OSGi Working Group

OSGi Compendium

Release 8.1

December 2022

Preface

Implementation Requirements

An implementation of a Specification: (i) must fully implement the Specification including all its required interfaces and functionality; (ii) must not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Specification. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Specification and must not be described as an implementation of the Specification. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. An implementation of a Specification must not claim to be a compatible implementation of the Specification unless it passes the Technology Compatibility Kit ("TCK") for the Specification.

Feedback

This specification can be downloaded from the OSGi Documentation web site:

`https://docs.osgi.org/specification/`

Comments about this specification can be raised at:

`https://github.com/osgi/osgi/issues`

Table of Contents

1 Introduction

This compendium contains the specifications of all current OSGi services.

1.1 Reader Level

This specification is written for the following audiences:

- Application developers
- Framework and system service developers (system developers)
- Architects

This specification assumes that the reader has at least one year of practical experience in writing Java programs. Experience with embedded systems and server-environments is a plus. Application developers must be aware that the OSGi environment is significantly more dynamic than traditional desktop or server environments.

System developers require a very deep understanding of Java. At least three years of Java coding experience in a system environment is recommended. A Framework implementation will use areas of Java that are not normally encountered in traditional applications. Detailed understanding is required of class loaders, garbage collection, Java 2 security, and Java native library loading.

Architects should focus on the introduction of each subject. This introduction contains a general overview of the subject, the requirements that influenced its design, and a short description of its operation as well as the entities that are used. The introductory sections require knowledge of Java concepts like classes and interfaces, but should not require coding experience.

Most of these specifications are equally applicable to application developers and system developers.

1.2 Version Information

This document is the Compendium Specification for the OSGi Compendium Release 8.1.

1.2.1 OSGi Core Release 8

This specification is based on OSGi Core Release 8. This specification can be downloaded from:

`https://docs.osgi.org/specification/`

1.2.2 Component Versions

Components in this specification have their own specification version, independent of this specification. The following table summarizes the packages and specification versions for the different subjects.

Table 1.1 Packages and versions

Item	Package	Version
100 *Remote Services*	-	Version 1.1
103 *Device Access Specification*	org.osgi.service.device	Version 1.1
104 *Configuration Admin Service Specification*	org.osgi.service.cm	Version 1.6
	org.osgi.service.cm.annotations	

Item	Package	Version
105 *Metatype Service Specification*	org.osgi.service.metatype	Version 1.4
	org.osgi.service.metatype.annotations	
106 *PreferencesService Specification*	org.osgi.service.prefs	Version 1.1
107 *User Admin Service Specification*	org.osgi.service.useradmin	Version 1.1
108 *Wire Admin Service Specification*	org.osgi.service.wireadmin	Version 1.0
111 *Device Service Specification for UPnP™ Technology*	org.osgi.service.upnp	Version 1.2
112 *Declarative Services Specification*	org.osgi.service.component	Version 1.5
	org.osgi.service.component.annotations	
	org.osgi.service.component.propertytypes	
	org.osgi.service.component.runtime	
	org.osgi.service.component.runtime.dto	
113 *Event Admin Service Specification*	org.osgi.service.event	Version 1.4
	org.osgi.service.event.annotations	
	org.osgi.service.event.propertytypes	
117 *Dmt Admin Service Specification*	org.osgi.service.dmt	Version 2.0
	org.osgi.service.dmt.notification	
	org.osgi.service.dmt.notification.spi	
	org.osgi.service.dmt.security	
	org.osgi.service.dmt.spi	
122 *Remote Service Admin Service Specification*	org.osgi.service.remoteserviceadmin	Version 1.1
	org.osgi.service.remoteserviceadmin.namespace	
123 *JTA Transaction Services Specification*	-	Version 1.0
125 *Data Service Specification for JDBC™ Technology*	org.osgi.service.jdbc	Version 1.1
126 *JNDI Services Specification*	org.osgi.service.jndi	Version 1.0
127 *JPA Service Specification*	org.osgi.service.jpa	Version 1.1
	org.osgi.service.jpa.annotations	
128 *Web Applications Specification*	-	Version 1.0
130 *Coordinator Service Specification*	org.osgi.service.coordinator	Version 1.0
131 *TR069 Connector Service Specification*	org.osgi.service.tr069todmt	Version 1.0
132 *Repository Service Specification*	org.osgi.service.repository	Version 1.1
133 *Service Loader Mediator Specification*	org.osgi.service.serviceloader	Version 1.0
135 *Common Namespaces Specification*	org.osgi.namespace.contract	Version 1.2
	org.osgi.namespace.extender	
	org.osgi.namespace.implementation	
	org.osgi.namespace.service	
	org.osgi.namespace.unresolvable	
137 *REST Management Service Specification*	org.osgi.service.rest	Version 1.0
	org.osgi.service.rest.client	

Item	**Package**	**Version**
138 *Asynchronous Service Specification*	org.osgi.service.async	Version 1.0
	org.osgi.service.async.delegate	
139 *Device Service Specification for EnOcean™ Technology*	org.osgi.service.enocean	Version 1.0
	org.osgi.service.enocean.descriptions	
140 *Jakarta Servlet Whiteboard*	org.osgi.service.servlet.whiteboard	Version 2.0
	org.osgi.service.servlet.whiteboard.annotations	
	org.osgi.service.servlet.whiteboard.propertytypes	
	org.osgi.service.servlet.context	
	org.osgi.service.servlet.runtime	
	org.osgi.service.servlet.runtime.dto	
141 *Device Abstraction Layer Specification*	org.osgi.service.dal	Version 1.0
142 *Device Abstraction Layer Functions Specification*	org.osgi.service.dal.functions	Version 1.0
	org.osgi.service.dal.functions.data	
143 *Network Interface Information Service Specification*	org.osgi.service.networkadapter	Version 1.0
144 *Resource Monitoring Specification*	org.osgi.service.resourcemonitoring	Version 1.0
	org.osgi.service.resourcemonitoring.monitor	
145 *USB Information Device Category Specification*	org.osgi.service.usbinfo	Version 1.0
146 *Serial Device Service Specification*	org.osgi.service.serial	Version 1.0
147 *Transaction Control Service Specification*	org.osgi.service.transaction.control	Version 1.0
	org.osgi.service.transaction.control.jdbc	
	org.osgi.service.transaction.control.jpa	
	org.osgi.service.transaction.control.recovery	
148 *Cluster Information Specification*	org.osgi.service.clusterinfo	Version 1.0
	org.osgi.service.clusterinfo.dto	
149 *Device Service Specification for ZigBee™ Technology*	org.osgi.service.zigbee	Version 1.0
	org.osgi.service.zigbee.descriptions	
	org.osgi.service.zigbee.descriptors	
	org.osgi.service.zigbee.types	
150 *Configurator Specification*	org.osgi.service.configurator	Version 1.0
	org.osgi.service.configurator.annotations	
	org.osgi.service.configurator.namespace	
151 *Jakarta RESTful Web Services Whiteboard*	org.osgi.service.jakartars.runtime	Version 2.0
	org.osgi.service.jakartars.runtime.dto	
	org.osgi.service.jakartars.whiteboard	
	org.osgi.service.jakartars.whiteboard.annotations	
	org.osgi.service.jakartars.whiteboard.propertytypes	
	org.osgi.service.jakartars.client	

Item	Package	Version
152 *CDI Integration Specification*	org.osgi.service.cdi	Version 1.0
	org.osgi.service.cdi.annotations	
	org.osgi.service.cdi.propertytypes	
	org.osgi.service.cdi.reference	
	org.osgi.service.cdi.runtime	
	org.osgi.service.cdi.runtime.dto	
	org.osgi.service.cdi.runtime.dto.template	
153 *Service Layer API for oneM2M™*	org.osgi.service.onem2m	Version 1.0
	org.osgi.service.onem2m.dto	
154 *Residential Device Management Tree Specification*	org.osgi.dmt.residential[1]	Version 1.0
155 *TR-157 Amendment 3 Software Module Guidelines*	-	Version 1.0
157 *Typed Event Service Specification*	org.osgi.service.typedevent	Version 1.0
	org.osgi.service.typedevent.annotations	
	org.osgi.service.typedevent.monitor	
	org.osgi.service.typedevent.propertytypes	
158 *Log Stream Provider Service Specification*	org.osgi.service.log.stream	Version 1.0
159 *Feature Service Specification*	org.osgi.service.feature	Version 1.0
702 *XML Parser Service Specification*	org.osgi.util.xml	Version 1.0
705 *Promises Specification*	org.osgi.util.promise	Version 1.3
	org.osgi.util.function	
706 *Push Stream Specification*	org.osgi.util.pushstream	Version 1.1
707 *Converter Specification*	org.osgi.util.converter	Version 1.0

When a component is represented in a bundle, a version attribute is needed in the declaration of the Import-Package or Export-Package manifest headers.

1.2.3 Notes

1. This is not a Java package but contains DMT Types.

1.3 References

[1] *OSGi Specifications*
https://docs.osgi.org/specification/

1.4 Changes

- Updated 705 *Promises Specification.*
- Updated 125 *Data Service Specification for JDBC™ Technology.*
- Updated 706 *Push Stream Specification.*

- 140 *Jakarta Servlet Whiteboard* updated for Jakarta EE and replaces the Http Whiteboard specification which is based upon the `javax`-namespace Servlet API. The old Http Service specification was also removed as it is based upon the old version 2.1 of the `javax`-namespace Servlet API.
- 151 *Jakarta RESTful Web Services Whiteboard* updated for Jakarta EE and replaces the JaxRS Whiteboard specification which is based upon the `javax`-namespace JAX-RS API.

100 Remote Services

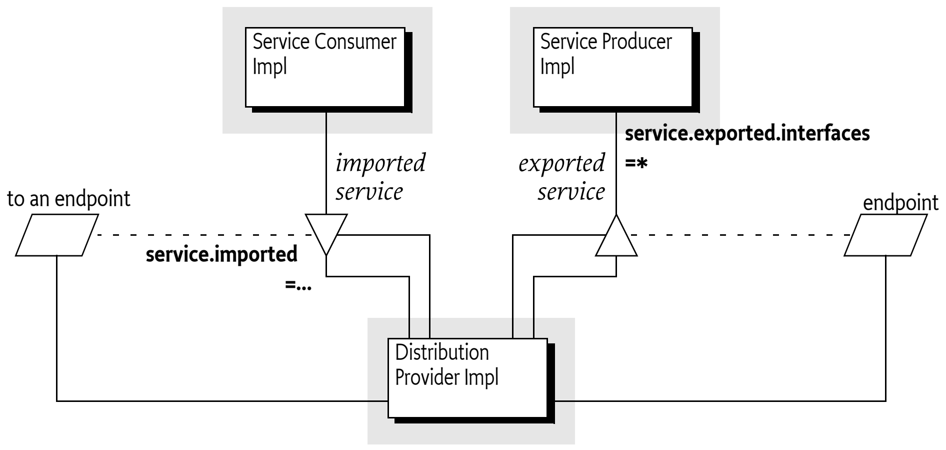
Version 1.1

The OSGi framework provides a *local* service registry for bundles to communicate through service objects, where a service is an object that one bundle registers and another bundle gets. A *distribution provider* can use this loose coupling between bundles to *export* a registered service by creating an *endpoint*. Vice versa, the distribution provider can create a *proxy* that accesses an endpoint and then registers this proxy as an *imported* service. A Framework can contain multiple distribution providers simultaneously, each independently importing and exporting services.

An endpoint is a communications access mechanisms to a service in another framework, a (web) service, another process, or a queue or topic destination, etc., requiring some protocol for communications. The constellation of the mapping between services and endpoints as well as their communication characteristics is called the *topology*. A common case for distribution providers is to be present on multiple frameworks importing and exporting services; effectively distributing the service registry.

The local architecture for remote services is depicted in Figure 100.1 on page 25.

Figure 100.1 *Architecture*



Local services imply in-VM call semantics. Many of these semantics cannot be supported over a communications connection, or require special configuration of the communications connection. It is therefore necessary to define a mechanism for bundles to convey their assumptions and requirements to the distribution provider. This chapter defines a number of service properties that a distribution provider can use to establish a topology while adhering to the given constraints.

100.1 The Fallacies

General abstractions for distributed systems have been tried before and often failed. Well known are the fallacies described in [1] *The Fallacies of Distributed Computing Explained*:

- The network is reliable

- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Most fallacies represent non-functional trade-offs that should be considered by administrators, their decisions can then be reflected in the topology. For example, in certain cases limited bandwidth is acceptable and the latency in a datacenter is near zero. However, the reliability fallacy is the hardest because it intrudes into the application code. If a communication channel is lost, the application code needs to take specific actions to recover from this failure.

This reliability aspect is also addressed with OSGi services because services are dynamic. Failures in the communications layer can be mapped to the unregistration of the imported service. OSGi bundles are already well aware of these dynamics, and a number of programming models have been developed to minimize the complexity of writing these dynamic applications.

100.2 Remote Service Properties

This section introduces a number of properties that participating bundles can use to convey information to the distribution provider according to this Remote Service specification.

The following table lists the properties that must be listed by a distribution provider.

Table 100.1 *Remote Service Properties registered by the Distribution Provider*

Service Property Name	Type	Description
`remote.configs.supported`	`String+`	Registered by the distribution provider on one of its services to indicate the supported configuration types. See *Configuration Types* on page 33 and *Dependencies* on page 36.
`remote.intents.supported`	`String+`	Registered by the distribution provider on one of its services to indicate the vocabulary of implemented intents. See *Dependencies* on page 36.
`service.imported`	`*`	Must be set by a distribution provider to any value when it registers the endpoint proxy as an imported service. A bundle can use this property to filter out imported services.
`service.imported.configs`	`String+`	The configuration information used to import this service, as described in `service.exported.configs`. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework. If multiple configuration types are listed in this property, then they must be synonyms for exactly the same remote endpoint that is used to export this service.

Service Property Name	Type	Description
service.intents	String+	A distribution provider must use this property to convey the combined intents of: • The exporting service, and • The intents that the exporting distribution provider adds. • The intents that the importing distribution provider adds.

The properties for bundles providing services to be exported or require services to be imported are listed alphabetically in the following table. The scenarios that these properties are used in are discussed in later sections.

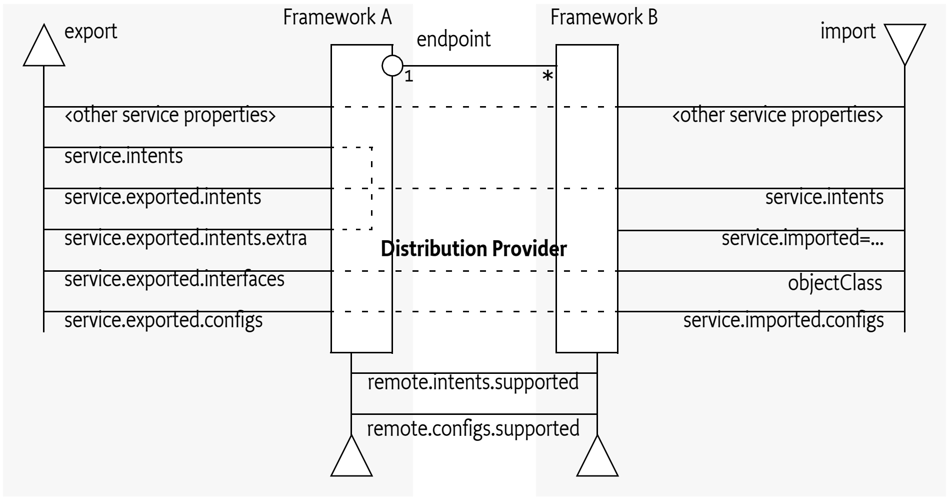
Table 100.2 *Remote Service Properties registered by Exporting bundles*

Service Property Name	Type	Description
service.exported.configs	String+	A list of configuration types that should be used to export the service. Each configuration type represents the configuration parameters for one or more Endpoints. A distribution provider should create endpoints for each configuration type that it supports. See *Configuration Types* on page 33 for more details. If this property is not set or empty a distribution provider is free to choose a default configuration type for the service.
service.exported.intents	String+	A list of *intents* that the distribution provider must implement to distribute the service. Intents listed in this property are reserved for intents that are critical for the code to function correctly, for example, ordering of messages. These intents should not be configurable. For more information about intents, see *Intents* on page 30. This property is optional.
service.exported.intents.extra	String+	This property is merged with the service.exported.intents property before the distribution provider interprets the listed intents; it has therefore the same semantics but the property should be configurable so the administrator can choose the intents based on the topology. Bundles should therefore make this property configurable, for example through the Configuration Admin service. See *Intents* on page 30. This property is optional. If absent or empty no specific intents are required.
service.exported.interfaces	String+	Setting this property marks this service for export. It defines the interfaces under which this service can be exported. This list must be a subset of the types listed in the objectClass service property. The single value of an asterisk ('*' \u002A) indicates all interfaces in the registration's objectClass property and ignore the classes. It is strongly recommended to only export interfaces and not concrete classes due to the complexity of creating proxies for some type of concrete classes. See *Registering a Service for Export* on page 28.

Service Property Name	Type	Description
service.intents	String+	A list of intents that this service implements. A distribution provider must use this property to convey the combined intents of: • The exporting service, and • The intents that the exporting distribution provider adds. • The intents that the importing distribution provider adds. To export a service, a distribution provider must expand any qualified intents to include those supported by the endpoint. This can be a subset of all known qualified intents. See *Intents* on page 30. This property is optional for registering bundles.
service.pid	String+	Services that are exported should have a service.pid property. The service.pid (PID) is a unique persistent identity for the service, the PID is defined in *Persistent Identifier (PID)* of *OSGi Core Release 8*. This property enables a distribution provider to associate persistent proprietary data with a service registration.

The properties and their treatment by the distribution provider is depicted in Figure 100.2.

Figure 100.2 *Distribution Service Properties*



100.2.1 Registering a Service for Export

A distribution provider should create one or more endpoints for an exported service when the following conditions are met:

- The service has the service property service.exported.interfaces set.
- All intents listed in service.exported.intents, service.exported.intents.extra and service.intents are part of the distributed provider's vocabulary
- None of the intents are mutually exclusive.
- The distribution provider can use the configuration types in service.exported.configs to create one or more endpoints.

The endpoint must at least implement all the intents that are listed in the service.exported.intents and service.exported.intents.extra properties.

The configuration types listed in the service.exported.configs can contain *alternatives* and/or *synonyms*. Alternatives describe different endpoints for the same service while a synonym describes a different configuration type for the same endpoint.

A distribution provider should create endpoints for each of the configuration types it supports; these configuration types should be alternatives. Synonyms are allowed.

If no configuration types are recognized, the distribution provider should create an endpoint with a default configuration type except when one of the listed configuration types is <<nodefault>>.

For more information about the configuration types, see further *Configuration Types* on page 33.

100.2.2 Getting an Imported Service

An imported service must be a normal service, there are therefore no special rules for getting it. An imported service has a number of additional properties that must be set by the distribution provider.

If the endpoint for an exported service is imported as an OSGi service in another framework, then the following properties must be treated as special.

- service.imported - Must be set to some value.
- service.intents - This must be the combination of the following:
  - The service.intents property on the exported service
  - The service.exported.intents and service.exported.intents.extra properties on the exported service
  - Any additional intents implemented by the distribution providers on both sides.
- service.imported.configs - Contains the configuration types that can be used to import this service. The types listed in this property must be *synonymous*, that is, they must refer to exactly the same endpoint that is exporting the service. See *Configuration Types* on page 33.
- service.exported.* - Properties starting with service.exported. must not be set on the imported service.
- service.exported.interfaces - This property must not be set, its content is reflected in the objectClass property.

All other *public* service properties (not starting with a full stop ('.' \u002E)) must be listed on the imported service if they use the basic service property types. If the service property cannot be communicated because, for example, it uses a type that can not be marshaled by the distribution provider then the distribution provider must ignore this property.

The service.imported property indicates that a service is an imported service. If this service property is set to any value, then the imported service is a proxy for an endpoint. If a bundle wants to filter out imported services, then it can add the following filter:

```
(&(!(service.imported=*)) <previousfilter>)
```

Distribution providers can also use the *Service Hook Service Specification* of *OSGi Core Release 8* to hide services from specific bundles.

100.2.3 On Demand Import

The Service Hooks Service Specification of *OSGi Core Release 8*, allows a distribution provider to detect when a bundle is listening for specific services. Bundles can request imported services with specific intents by building an appropriate filter. The distribution provider can use this information to import a service on demand.

The following example creates a Service Tracker that is interested in an imported service.

```
Filter f = context.createFilter(
        "(&(objectClasss=com.acme.Foo)"
    +   "(service.intents=confidentiality))"
);
ServiceTracker tracker =
    new ServiceTracker(context, f, null );
tracker.open();
```

Such a Service Tracker will inform the Listener Hook and will give it the filter expression. If the distribution provider has registered such a hook, it will be informed about the need for an imported com.acme.Foo service that has a confidentiality intent. It can then use some proprietary means to find a service to import that matches the given object class and intent.

How the distribution provider finds an appropriate endpoint is out of scope for this specification.

100.3 Intents

An intent is a name for an abstract distribution capability. An intent can be *implemented* by a service; this can then be reflected in the service.intents property. An intent can also *constrain* the possible communication mechanisms that a distribution provider can choose to distribute a service. This is reflected in the service.exported.intents and service.exported.intents.extra properties.

The purpose of the intents is to have a *vocabulary* that is shared between distribution aware bundles and the distribution provider. This vocabulary allows the bundles to express constraints on the export of their services as well as providing information on what intents are implemented by a service.

Intents have the following syntax

```
intent  ::= token ( '.' token )?
```

Qualified intents use a full stop ('.' \u002E) to separate the intent from the qualifier. A qualifier provides additional details, however, it implies its prefix. For example:

```
confidentiality.message
```

This example, can be *expanded* into confidentiality and confidentiality.message. Qualified intents can be used to provide additional details how an intent is achieved. However, a Distribution Provider must expand any qualified intents to include those supported by the endpoint. This can be a subset of all known qualified intents.

The concept of intents is derived from the [3] *SCA Policy Framework specification.* When designing a vocabulary for a distribution provider it is recommended to closely follow the vocabulary of intents defined in the SCA Policy Framework.

100.3.1 Basic Remote Services: osgi.basic

Remote Services implementations have a large amount of freedom. For example, they may use any mechanism that they choose to transmit data between the caller of the remote service and the provider of the service. This freedom means that there can be a large variation in the behaviors supported by different Remote Services implementations.

The purpose of the osgi.basic intent is to provide a common set of rules that can be relied upon when exporting a simple remote service. This includes rules about the service interface, including supported parameter and return types, as well as a means of configuring a timeout for remote invocations.

100.3.1.1 Minimum Supported Service Signature

Remote Services implementations which offer the osgi.basic intent must support remote services which advertise a single Java interface containing zero or more methods.

The following types must be supported as declared parameters or returns from methods on the remote service:

- Primitive values
- The OSGi scalar types, OSGi Version objects, Java enums, and types which conform to the OSGi DTO rules as described in the OSGi core specification. In the rest of this section these will be known as the *basic types*.
- Arrays of primitive values or the basic types
- Lists, Collections or Iterables of the basic types, however the implementation of the collection may not be preserved in transit. For example a LinkedList may be converted to an ArrayList.
- Sets of the OSGi basic types where equals is used to determine identity. SortedSet is not required to be supported due to the difficulties associated with serializing comparators. The implementation of the set may not be preserved in transit. For example a LinkedHashSet may be converted to a HashSet.
- Maps where the keys and values are the OSGi basic types, and equals is used to determine identity for the keys. SortedMap is not required to be supported due to the difficulties associated with serializing comparators. The implementation of the map may not be preserved in transit. For example a LinkedHashMap may be converted to a HashMap.
- Methods with no arguments, and methods with a void return

100.3.1.2 Remote Invocation Timeout

The implementation of a Remote Services provider is entirely opaque. In many cases there will be no feedback mechanism if the remote call hangs, or if the remote node fails. The local client must therefore decide at what point to fail after a certain amount of time has elapsed.

A single Remote Services implementation must be able to handle a wide variety of different remote service invocations across many services, therefore it is difficult to identify a sensible timeout for the remote service invocation. Some calls may be quick, and so a ten second timeout is desirable for rapid failure detection, other calls may be long-running, and a two minute timeout too short. The remote service must therefore be able to declare its own timeout.

To declare a timeout the remoteable service may provide a service property osgi.basic.timeout which provides a timeout value in milliseconds. The value may be declared as a String or as a Number, which will be converted into a Long. The timeout value is used to limit the maximum time for which a remote service client will be blocked waiting for a response. The same timeout value applies to all methods on the service. In the event that the invocation reaches the timeout value the client must fail the method call with a ServiceException with its type set to REMOTE.

100.3.2 Asynchronous Remote Services: osgi.async

Some service invocations operate asynchronously, returning quickly and continuing to process in the background. For void methods with no completion notifications this is simple to achieve remotely, but more useful scenarios are difficult to support without using higher-level abstractions to represent the eventual result.

The purpose of the osgi.async intent is to provide a common set of rules that can be relied upon for remote services which return types representing an asynchronously executing method.

The osgi.async intent makes no guarantees about the service interface(s) or method parameters supported by the remote services implementation. It is therefore recommended that it be used in conjunction with another intent, such as the osgi.basic intent.

100.3.2.1 Supported Return Types

Asynchronous returns are implemented using a holder type. The holder represents the state of the asynchronous execution, and can be queried for its completion state. When the execution is complete the holder can be queried for the result of the execution, or for its failure.

The following holder types must be supported as return types from methods on the remote service:

- `org.osgi.util.promise.Promise`
- `java.util.concurrent.Future`
- `java.util.concurrent.CompletionStage`
- `java.util.concurrent.CompletableFuture`

The full set of supported types for the eventual return value encapsulated by the holder object are not defined by the `osgi.async` intent. Instead the full set of supported types can be inferred from the other supported intents supported by the Remote Services implementation. For example the osgi.basic intent would ensure support for a return value of `Promise<List<String>>`

100.3.2.2 Asynchronous Failures

If an asynchronous remote execution fails then the holder type must be failed with the same exception that would have been thrown in a synchronous call.

The reason for the failure may be as a result of a failure in communications, a timeout, or because the remote invocation resulted in an exception

100.3.3 Confidential Remote Services: osgi.confidential

The `osgi.confidential` intent can be used to state that the remote service communications must only be readable by the intended recipient, for example, through the use of TLS-based transport encryption.

If a Remote Services implementation does not support confidential communications, or is not configured as such, it must not expose the service remotely.

100.3.4 Private Remote Services: osgi.private

In many deployment scenarios, including cloud, embedded or IoT deployments, hosts may be accessible via a public network and via a private network. In such cases hosts will have multiple IP addresses to separate public network access from private network access. Private IP addresses normally in one of the following blocks: `10.0.0.0/8`, `172.16.0.0/12` or `192.168.0.0/16`.

In many cases it is desirable to expose remote services only on the private network so that these services cannot be accessed from the outside world. This is especially useful if this service is used as a microservice within a larger application. The `osgi.private` intent can be specified for this purpose.

If the `osgi.private` intent is required on the remote service, it will only be exposed as a remote service on a private network on the host. If the host does not support a private IP address or if the Remote Services implementation does not have the information to decide whether a host IP is private, the service should not be exposed.

100.4 General Usage

100.4.1 Call by Value

Normal service semantics are call-by-reference. An object passed as an argument in a service call is a direct reference to that object. Any changes to this object will be shared on both sides of the service registry.

Distributed services are different. Arguments are normally passed by value, which means that a copy is sent to the remote system, changes to this value are not reflected in the originating framework. When using distributed services, call-by-value should always be assumed by all participants in the distribution chain.

100.4.2 Data Fencing

Services are syntactically defined by their Java interfaces. When exposing a service over a remote protocol, typically such an interface is mapped to a protocol-specific interface definition. For example, in CORBA the Java interfaces would be converted to a corresponding IDL definition. This mapping does not always result in a complete solution.

Therefore, for many practical distributed applications it will be necessary to constrain the possible usage of data types in service interfaces. A distribution provider must at least support interfaces (not classes) that only use the basic types as defined for the service properties. These are the primitive types and their wrappers as well as arrays and collections. See *Filter Syntax* of *OSGi Core Release 8* for a list of service property types.

Distribution providers will in general provide a richer set of types that can be distributed.

100.4.3 Remote Services Life Cycle

A distributed service must closely track any modifications on the corresponding service registration. If service properties are modified, these modifications should be propagated to the distributed service and associated service proxies. If the exported service is unregistered, the endpoint must be withdrawn as soon as possible and any imported service proxies unregistered.

100.4.4 Runtime

An imported service is just like any other service and can be used as such. However, certain non-functional characteristics of this service can differ significantly from what is normal for an in-VM object call. Many of these characteristics can be mapped to the normal service operations. That is, if the connection fails in any way, the service can be unregistered. According to the standard OSGi contract, this means that the users of that service must perform the appropriate cleanup to prevent stale references.

100.4.5 Exceptions

It is impossible to guarantee that a service is not used when it is no longer valid. Even with the synchronous callbacks from the Service Listeners, there is always a finite window where a service can be used while the underlying implementation has failed. In a distributed environment, this window can actually be quite large for an imported service.

Such failure situations must be exposed to the application code that uses a failing imported service. In these occasions, the distribution provider must notify the application by throwing a Service Exception, or subclass thereof, with the reason `REMOTE`. The Service Exception is a Runtime Exception, it can be handled higher up in the call chain. The cause of this Service Exception must be the Exception that caused the problem.

A distribution provider should log any problems with the communications layer to the Log Service, if available.

100.5 Configuration Types

An exported service can have a `service.exported.configs` service property. This property lists configuration types for endpoints that are provided for this service. Each type provides a specification that defines how the configuration data for one or more endpoints is provided. For example, a hypothetical configuration type could use a service property to hold a URL for the RMI naming registry.

Configuration types that are not defined by the OSGi Working Group should use a name that follows the reverse capabilities domain name scheme defined in [4] *Java Language Specification* for Java packages. For example, com.acme.wsdl would be the proprietary way for the ACME company to specify a WSDL configuration type.

100.5.1 Configuration Type Properties

The service.exported.configs and service.imported.configs use the configuration types in very different ways. That is, the service.imported.configs property is not a copy of the service.exported.configs as the name might seem to imply.

An exporting service can list its desired configuration types in the service.exported.configs property. This property is potentially seen and interpreted by multiple distribution providers. Each of these providers can independently create endpoints from the configuration types. In principle, the service.exported.configs lists *alternatives* for a single distribution provider and can list *synonyms* to support alternative distribution providers. If only one of the synonyms is useful, there is an implicit assumption that when the service is exported, only one of the synonyms should be supported by the installed distribution providers. If it is detected that this assumption is violated, then an error should be logged and the conflicting configuration is further ignored.

The interplay of synonyms and alternatives is depicted in Table 100.3. In this table, the first columns on the left list different combinations of the configuration types in the service.exported.configs property. The next two columns list two distribution providers that each support an overlapping set of configuration types. The x's in this table indicate if a configuration type or distribution provider is active in a line. The description then outlines the issues, if any. It is assumed in this table that hypothetical configuration types net.rmi and com.rmix map to an identical endpoint, just like net.soap and net.soapx.

Table 100.3 Synonyms and Alternatives in Exported Configurations

service.exported. configs					**Distribution Provider A**	**Distribution Provider B**	**Description**
net.rmi	com.rmix	net.soap	com.soapx	<<no default>>	Supports: net.rmi com.rmix com.soapx	Supports: net.rmi net.soap	
x		x			x		*OK*, A will create an endpoint for the RMI and SOAP alternatives.
x					x	x	*Configuration error*. There is a clash for net.rmi because A and B can both create an endpoint for the same configuration. It is likely that one will fail.
		x	x		x		*OK*, exported on com.soapx by A, the net.soap is ignored.
		x	x		x	x	*Synonym error* because A and B export to same SOAP endpoint, it is likely that one will fail.
	x	x			x	x	*OK*, two alternative endpoints over RMI (by A) and SOAP (by B) are created. This is a typical use case.
		x	x		x		*OK*. Synonyms are used to allow frameworks that have either A or B installed. In this case A exports over SOAP.
		x	x			x	*OK*. Synonyms are used to allow frameworks that have either A or B installed. In this case B exports.
					x		*OK*. A creates an endpoint with default configuration type.

service.exported. configs	**Distribution Provider A**	**Distribution Provider B**	**Description**
	x	x	*OK*. Both A and B each create an endpoint with their default configuration type.
x	x		*OK*. No endpoint is created.
x x		x	Provider B does not recognize the configuration types it should therefore use a default configuration type.

To summarize, the following rules apply for a single distribution provider:

- Only configuration types that are supported by this distribution provider must be used. All other configuration types must be ignored.
- All of the supported configuration types must be *alternatives*, that is, they must map to different endpoints. Synonyms for the same distribution provider should be logged as errors.
- If a configuration type results in an endpoint that is already in use, then an error should be logged. It is likely then that another distribution provider already had created that endpoint.

An export of a service can therefore result in multiple endpoints being created. For example, a service can be exported over RMI as well as SOAP. Creating an endpoint can fail, in that case the distribution provider must log this information in the Log Service, if available, and not export the service to that endpoint. Such a failure can, for example, occur when two configuration types are synonym and multiple distribution providers are installed that supporting this type.

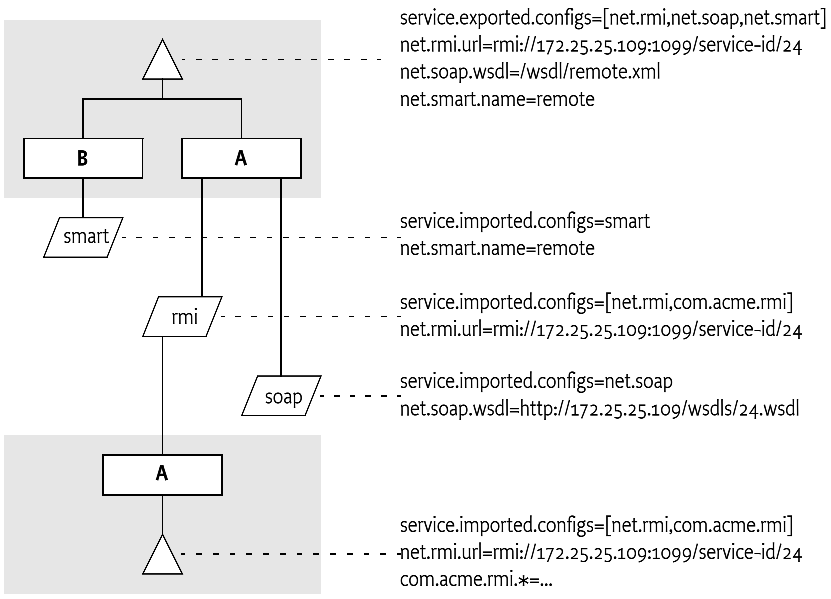
On the importing side, the `service.imported.configs` property lists configuration types that must refer to the same endpoint. That is, it can list alternative configuration types for this endpoint but all configuration types must result in the same endpoint.

For example, there are two distribution providers installed at the exporting and importing frameworks. Distribution provider A supports the hypothetical configuration type `net.rmi` and `net.soap`. Distribution provider B supports the hypothetical configuration type `net.smart`. A service is registered that list all three of those configuration types.

Distribution provider A will create two endpoints, one for RMI and one for SOAP. Distribution provider B will create one endpoint for the smart protocol. The distribution provider A knows how to create the configuration data for the `com.acme.rmi` configuration type as well and can therefore create a synonymous description of the endpoint in that configuration type. It will therefore set the imported configuration type for the RMI endpoint to:

```
service.imported.configs = net.rmi, com.acme.rmi
net.rmi.url = rmi://172.25.25.109:1099/service-id/24
com.acme.rmi.address = 172.25.25.109
com.acme.rmi.port = 1099
com.acme.rmi.path = service-id/24
```

Figure 100.3 *Relation between imported and exported configuration types*


service.exported.configs=[net.rmi,net.soap,net.smart]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24
net.soap.wsdl=/wsdl/remote.xml
net.smart.name=remote
B
A
smart
service.imported.configs=smart
net.smart.name=remote
rmi
service.imported.configs=[net.rmi,com.acme.rmi]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24
soap
service.imported.configs=net.soap
net.soap.wsdl=http://172.25.25.109/wsdls/24.wsdl
A
service.imported.configs=[net.rmi,com.acme.rmi]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24
com.acme.rmi.*=...

100.5.2 Dependencies

A bundle that uses a configuration type has an implicit dependency on the distribution provider. To make this dependency explicit, the distribution provider must register a service with the following properties:

- `remote.intents.supported` - (`String+`) The vocabulary of the given distribution provider.
- `remote.configs.supported` - (`String+`) The configuration types that are implemented by the distribution provider.

A bundle that depends on the availability of specific intents or configuration types can create a service dependency on an anonymous service with the given properties. The following filter is an example of depending on a hypothetical `net.rmi` configuration type:

```
(remote.configs.supported=net.rmi)
```

100.6 Security

The distribution provider will be required to invoke methods on any exported service. This implies that it must have the combined set of permissions of all methods it can call. It also implies that the distribution provider is responsible for ensuring that a bundle that calls an imported service is not granted additional permissions through the fact that the distribution provider will call the exported service, not the original invoker.

The actual mechanism to ensure that bundles can get additional permissions through the distribution is out of scope for this specification. However, distribution providers should provide mechanisms to limit the set of available permissions for a remote invocation, preferably on a small granularity basis.

One possible means is to use the `getAccessControlContext` method on the Conditional Permission Admin service to get an Access Control Context that is used in a `doPrivileged` block where the invocation takes place. The `getAccessControlContext` method takes a list of signers which could represent the remote bundles that cause an invocation. How these are authenticated is up to the distribution provider.

A distribution provider is a potential attack point for intruders. Great care should be taken to properly setup the permissions or topology in an environment that requires security.

100.6.1 Limiting Exports and Imports

Service registration and getting services is controlled through the ServicePermission class. This permission supports a filter based constructor that can assert service properties. This facility can be used to limit bundles from being able to register exported services or get imported services if they are combined with Conditional Permission Admin's ALLOW facility. The following example shows how all bundles except from www.acme.com are denied the registration and getting of distributed services.

```
DENY {
   [...BundleLocationCondition("http://www.acme.com/*" "!")]
   (...ServicePermission "(service.imported=*)" "GET" )
   (...ServicePermission "(service.exported.interfaces=*)"
                              "REGISTER" )
}
```

100.7 References

[1] *The Fallacies of Distributed Computing Explained*
https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

[2] *Service Component Architecture (SCA)*
https://www.osoa.org/

[3] *SCA Policy Framework specification*
https://www.osoa.org/

[4] *Java Language Specification*
https://docs.oracle.com/javase/specs/

103 Device Access Specification

Version 1.1

103.1 Introduction

A Framework is a meeting point for services and devices from many different vendors: a meeting point where users add and cancel service subscriptions, newly installed services find their corresponding input and output devices, and device drivers connect to their hardware.

In an OSGi Framework, these activities will dynamically take place while the Framework is running. Technologies such as USB and IEEE 1394 explicitly support plugging and unplugging devices at any time, and wireless technologies are even more dynamic.

This flexibility makes it hard to configure all aspects of an OSGi Framework, particularly those relating to devices. When all of the possible services and device requirements are factored in, each OSGi Framework will be unique. Therefore, automated mechanisms are needed that can be extended and customized, in order to minimize the configuration needs of the OSGi environment.

The Device Access specification supports the coordination of automatic detection and attachment of existing devices on an OSGi Framework, facilitates hot-plugging and -unplugging of new devices, and downloads and installs device drivers on demand.

This specification, however, deliberately does not prescribe any particular device or network technology, and mentioned technologies are used as examples only. Nor does it specify a particular device discovery method. Rather, this specification focuses on the attachment of devices supplied by different vendors. It emphasizes the development of standardized device interfaces to be defined in device categories, although no such device categories are defined in this specification.

103.1.1 Essentials

- *Embedded Devices* - OSGi bundles will likely run in embedded devices. This environment implies limited possibility for user interaction, and low-end devices will probably have resource limitations.
- *Remote Administration* - OSGi environments must support administration by a remote service provider.
- *Vendor Neutrality* - OSGi-compliant driver bundles will be supplied by different vendors; each driver bundle must be well-defined, documented, and replaceable.
- *Continuous Operation* - OSGi environments will be running for extended periods without being restarted, possibly continuously, requiring stable operation and stable resource consumption.
- *Dynamic Updates* - As much as possible, driver bundles must be individually replaceable without affecting unrelated bundles. In particular, the process of updating a bundle should not require a restart of the whole OSGi Framework or disrupt operation of connected devices.

A number of requirements must be satisfied by Device Access implementations in order for them to be OSGi-compliant. Implementations must support the following capabilities:

- *Hot-Plugging* - Plugging and unplugging of devices at any time if the underlying hardware and drivers allow it.
- *Legacy Systems* - Device technologies which do not implement the automatic detection of plugged and unplugged devices.

- *Dynamic Device Driver Loading* - Loading new driver bundles on demand with no prior device-specific knowledge of the Device service.
- *Multiple Device Representations* - Devices to be accessed from multiple levels of abstraction.
- *Deep Trees* - Connections of devices in a tree of mixed network technologies of arbitrary depth.
- *Topology Independence* - Separation of the interfaces of a device from where and how it is attached.
- *Complex Devices* - Multifunction devices and devices that have multiple configurations.

103.1.2 Operation

This specification defines the behavior of a device manager (which is *not* a service as might be expected). This device manager detects registration of Device services and is responsible for associating these devices with an appropriate Driver service. These tasks are done with the help of Driver Locator services and the Driver Selector service that allow a device manager to find a Driver bundle and install it.

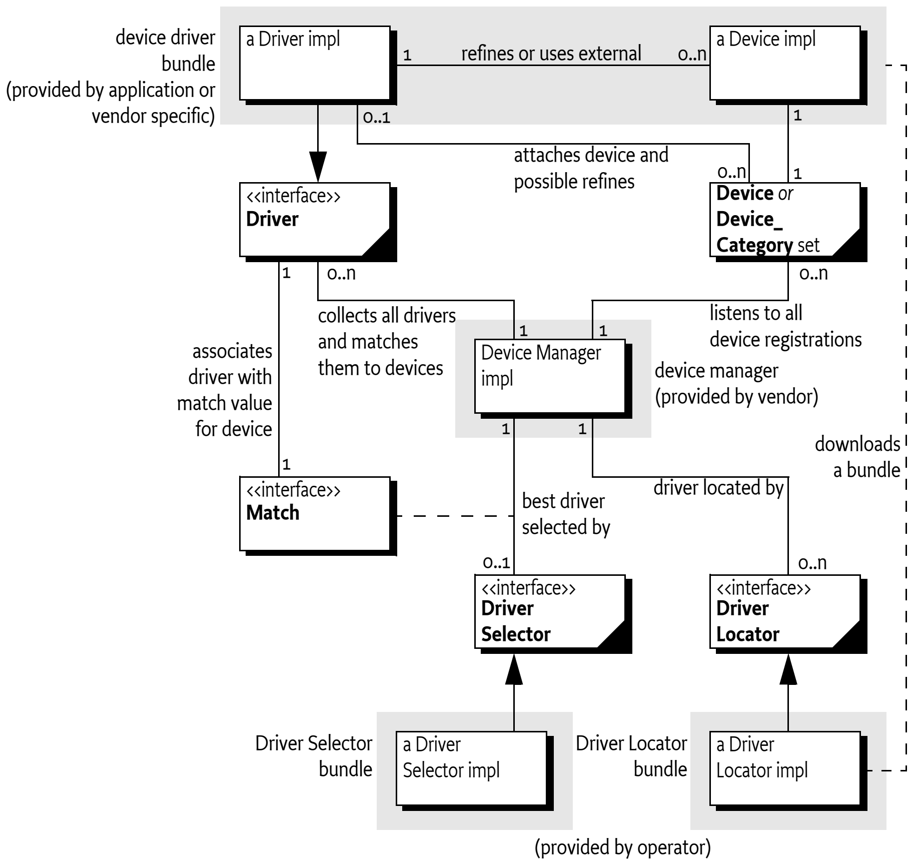
103.1.3 Entities

The main entities of the Device Access specification are:

- *Device Manager* - The bundle that controls the initiation of the attachment process behind the scenes.
- *Device Category* - Defines how a Driver service and a Device service can cooperate.
- *Driver* - Competes for attaching Device services of its recognized device category. See *Driver Services* on page 45.
- *Device* - A representation of a physical device or other entity that can be attached by a Driver service. See *Device Services* on page 41.
- *DriverLocator* - Assists in locating bundles that provide a Driver service. See *Driver Locator Service* on page 51.
- *DriverSelector* - Assists in selecting which Driver service is best suited to a Device service. See *The Driver Selector Service* on page 54.

Figure 103.1 show the classes and their relationships.

Figure 103.1 Device Access Class Overview


device driver bundle (provided by application or vendor specific)
a Driver impl
1
refines or uses external
0..n
a Device impl
0..1
1
attaches device and possible refines
0..n
1
<<interface>> Driver
Device or Device_ Category set
1
0..n
0..n
collects all drivers and matches them to devices
listens to all device registrations
associates driver with match value for device
1
1
Device Manager impl
device manager (provided by vendor)
1
1
downloads a bundle
1
<<interface>> Match
best driver selected by
driver located by
0..1
0..n
<<interface>> Driver Selector
<<interface>> Driver Locator
Driver Selector bundle
a Driver Selector impl
Driver Locator bundle
a Driver Locator impl
(provided by operator)

103.2 Device Services

A Device service represents some form of a device. It can represent a hardware device, but that is not a requirement. Device services differ widely: some represent individual physical devices and others represent complete networks. Several Device services can even simultaneously represent the same physical device at different levels of abstraction. For example:

- A USB network.
- A device attached on the USB network.
- The same device recognized as a USB to Ethernet bridge.
- A device discovered on the Ethernet using Salutation.
- The same device recognized as a simple printer.
- The same printer refined to a PostScript printer.

A device can also be represented in different ways. For example, a USB mouse can be considered as:

- A USB device which delivers information over the USB bus.
- A mouse device which delivers x and y coordinates and information about the state of its buttons.

Each representation has specific implications:

- That a particular device is a mouse is irrelevant to an application which provides management of USB devices.
- That a mouse is attached to a USB bus or a serial port would be inconsequential to applications that respond to mouse-like input.

Device services must belong to a defined *device category*, or else they can implement a generic service which models a particular device, independent of its underlying technology. Examples of this type of implementation could be Sensor or Actuator services.

A device category specifies the methods for communicating with a Device service, and enables interoperability between bundles that are based on the same underlying technology. Generic Device services will allow interoperability between bundles that are not coupled to specific device technologies.

For example, a device category is required for the USB, so that Driver bundles can be written that communicate to the devices that are attached to the USB. If a printer is attached, it should also be available as a generic Printer service defined in a Printer service specification, indistinguishable from a Printer service attached to a parallel port. Generic categories, such as a Printer service, should also be described in a Device Category.

It is expected that most Device service objects will actually represent a physical device in some form, but that is not a requirement of this specification. A Device service is represented as a normal service in the OSGi Framework and all coordination and activities are performed upon Framework services. This specification does not limit a bundle developer from using Framework mechanisms for services that are not related to physical devices.

103.2.1 Device Service Registration

A Device service is defined as a normal service registered with the Framework that either:

- Registers a service object under the interface `org.osgi.service.Device` with the Framework, or
- Sets the `DEVICE_CATEGORY` property in the registration. The value of `DEVICE_CATEGORY` is an array of `String` objects of all the device categories that the device belongs to. These strings are defined in the associated device category.

If this document mentions a Device service, it is meant to refer to services registered with the name `org.osgi.service.device.Device` *or* services registered with the `DEVICE_CATEGORY` property set.

When a Device service is registered, additional properties may be set that describe the device to the device manager and potentially to the end users. The following properties have their semantics defined in this specification:

- `DEVICE_CATEGORY` - A marker property indicating that this service must be regarded as a Device service by the device manager. Its value is of type `String[]`, and its meaning is defined in the associated device category specification.
- `DEVICE_DESCRIPTION` - Describes the device to an end user. Its value is of type `String`.
- `DEVICE_SERIAL` - A unique serial number for this device. If the device hardware contains a serial number, the driver bundle is encouraged to specify it as this property. Different Device services representing the same physical hardware at different abstraction levels should set the same `DEVICE_SERIAL`, thus simplifying identification. Its value is of type `String`.
- `service.pid` - Service Persistent ID (PID), defined in `org.osgi.framework.Constants`. Device services should set this property. It must be unique among all registered services. Even different abstraction levels of the same device must use different PIDs. The service PIDs must be reproducible, so that every time the same hardware is plugged in, the same PIDs are used.

103.2.2 Device Service Attachment

When a Device service is registered with the Framework, the device manager is responsible for finding a suitable Driver service and instructing it to attach to the newly registered Device service. The

Device service itself is passive: it only registers a Device service with the Framework and then waits until it is called.

The actual communication with the underlying physical device is not defined in the `Device` interface because it differs significantly between different types of devices. The Driver service is responsible for attaching the device in a device type-specific manner. The rules and interfaces for this process must be defined in the appropriate device category.

If the device manager is unable to find a suitable Driver service, the Device service remains unattached. In that case, if the service object implements the `Device` interface, it must receive a call to the `noDriverFound()` method. The Device service can wait until a new driver is installed, or it can unregister and attempt to register again with different properties that describe a more generic device or try a different configuration.

103.2.2.1 Idle Device Service

The main purpose of the device manager is to try to attach drivers to idle devices. For this purpose, a Device service is considered *idle* if no bundle that itself has registered a Driver service is using the Device service.

103.2.2.2 Device Service Unregistration

When a Device service is unregistered, no immediate action is required by the device manager. The normal service of unregistering events, provided by the Framework, takes care of propagating the unregistration information to affected drivers. Drivers must take the appropriate action to release this Device service and perform any necessary cleanup, as described in their device category specification.

The device manager may, however, take a device unregistration as an indication that driver bundles may have become idle and are thus eligible for removal. It is therefore important for Device services to unregister their service object when the underlying entity becomes unavailable.

103.3 Device Category Specifications

A device category specifies the rules and interfaces needed for the communication between a Device service and a Driver service. Only Device services and Driver services of the same device category can communicate and cooperate.

The Device Access service specification is limited to the attachment of Device services by Driver services, and does *not* enumerate different device categories.

Other specifications must specify a number of device categories before this specification can be made operational. Without a set of defined device categories, no interoperability can be achieved.

Device categories are related to a specific device technology, such as USB, IEEE 1394, JINI, UPnP, Salutation, CEBus, Lonworks, and others. The purpose of a device category specification is to make all Device services of that category conform to an agreed interface, so that, for example, a USB Driver service of vendor A can control Device services from vendor B attached to a USB bus.

This specification is limited to defining the guidelines for device category definitions only. Device categories may be defined by the OSGi organization or by external specification bodies - for example, when these bodies are associated with a specific device technology.

103.3.1 Device Category Guidelines

A device category definition comprises the following elements:

- An interface that all devices belonging to this category must implement. This interface should lay out the rules of how to communicate with the underlying device. The specification body may define its own device interfaces (or classes) or leverage existing ones. For example, a serial port device category could use the javax.comm.SerialPort interface which is defined in [1] *Java Communications API*.

When registering a device belonging to this category with the Framework, the interface or class name for this category must be included in the registration.

- A set of service registration properties, their data types, and semantics, each of which must be declared as either MANDATORY or OPTIONAL for this device category.
- A range of match values specific to this device category. Matching is explained later in *The Device Attachment Algorithm* on page 55.

103.3.2 Sample Device Category Specification

The following is a partial example of a fictitious device category:

```
public interface /* com.acme.widget.*/ WidgetDevice{
    int MATCH_SERIAL                   = 10;
    int MATCH_VERSION                  =  8;
    int MATCH_MODEL                    =  6;
    int MATCH_MAKE                     =  4;
    int MATCH_CLASS                    =  2;
    void sendPacket( byte [] data );
    byte [] receivePacket( long timeout );
}
```

Devices in this category must implement the interface com.acme.widget.WidgetDevice to receive attachments from Driver services in this category.

Device properties for this fictitious category are defined in the following table.

Table 103.1 Example Device Category Properties, M=Mandatory, O=Optional

Property name	M/O	Type	Value
DEVICE_CATEGORY	M	String[]	{"Widget"}
com.acme.class	M	String	A class description of this device. For example "audio", "video", "serial", etc. An actual device category specification should contain an exhaustive list and define a process to add new classes.
com.acme.model	M	String	A definition of the model. This is usually vendor specific. For example "Mouse".
com.acme.manufacturer	M	String	Manufacturer of this device, for example "ACME Widget Division".
com.acme.revision	O	String	Revision number. For example, "42".
com.acme.serial	O	String	A serial number. For example "SN6751293-12-2112/A".

103.3.3 Match Example

Driver services and Device services are connected via a matching process that is explained in *The Device Attachment Algorithm* on page 55. The Driver service plays a pivotal role in this matching process. It must inspect the Device service (from its ServiceReference object) that has just been registered and decide if it potentially could cooperate with this Device service.

It must be able to answer a value indicating the quality of the match. The scale of this match value must be defined in the device category so as to allow Driver services to match on a fair basis. The scale must start at least at 1 and go upwards.

Driver services for this sample device category must return one of the match codes defined in the com.acme.widget.WidgetDevice interface or Device.MATCH_NONE if the Device service is not recognized. The device category must define the exact rules for the match codes in the device category specification. In this example, a small range from 2 to 10 (MATCH_NONE is 0) is defined for WidgetDevice devices. They are named in the WidgetDevice interface for convenience and have the following semantics.

Table 103.2 *Sample Device Category Match Scale*

Match name	Value	Description
MATCH_SERIAL	10	An exact match, including the serial number.
MATCH_VERSION	8	Matches the right class, make model, and version.
MATCH_MODEL	6	Matches the right class and make model.
MATCH_MAKE	4	Matches the make.
MATCH_CLASS	2	Only matches the class.

A Driver service should use the constants to return when it decides how closely the Device service matches its suitability. For example, if it matches the exact serial number, it should return MATCH_SERIAL.

103.4 Driver Services

A Driver service is responsible for attaching to suitable Device services under control of the device manager. Before it can attach a Device service, however, it must compete with other Driver services for control.

If a Driver service wins the competition, it must attach the device in a device category-specific way. After that, it can perform its intended functionality. This functionality is not defined here nor in the device category; this specification only describes the behavior of the Device service, not how the Driver service uses it to implement its intended functionality. A Driver service may register one or more new Device services of another device category or a generic service which models a more refined form of the device.

Both refined Device services as well as generic services should be defined in a Device Category. See *Device Category Specifications* on page 43.

103.4.1 Driver Bundles

A Driver service is, like *all* services, implemented in a bundle, and is recognized by the device manager by registering one or more Driver service objects with the Framework.

Such bundles containing one or more Driver services are called *driver bundles*. The device manager must be aware of the fact that the cardinality of the relationship between bundles and Driver services is 1:1...*.

A driver bundle must register *at least* one Driver service in its BundleActivator.start implementation.

103.4.2 Driver Taxonomy

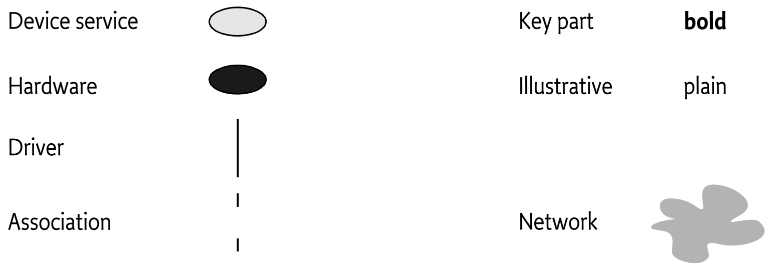
Device Drivers may belong to one of the following categories:

- Base Drivers (Discovery, Pure Discovery and Normal)
- Refining Drivers
- Network Drivers

- Composite Drivers
- Referring Drivers
- Bridging Drivers
- Multiplexing Drivers
- Pure Consuming Drivers

This list is not definitive, and a Driver service is not required to fit into one of these categories. The purpose of this taxonomy is to show the different topologies that have been considered for the Device Access service specification.

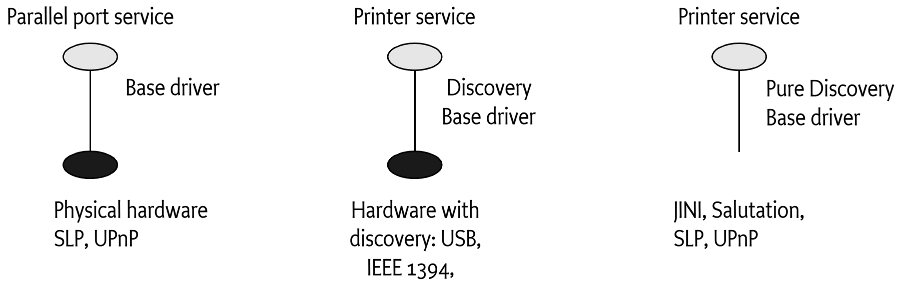
Figure 103.2 *Legend for Device Driver Services Taxonomy*



103.4.2.1 Base Drivers

The first category of device drivers are called *base drivers* because they provide the lowest-level representation of a physical device. The distinguishing factor is that they are not registered as Driver services because they do not have to compete for access to their underlying technology.

Figure 103.3 *Base Driver Types*



Base drivers discover physical devices using code not specified here (for example, through notifications from a device driver in native code) and then register corresponding Device services.

When the hardware supports a discovery mechanism and reports a physical device, a Device service is then registered. Drivers supporting a discovery mechanism are called *discovery base drivers*.

An example of a discovery base driver is a USB driver. Discovered USB devices are registered with the Framework as a generic USB Device service. The USB specification (see [2] *USB Specification*) defines a tightly integrated discovery method. Further, devices are individually addressed; no provision exists for broadcasting a message to all devices attached to the USB bus. Therefore, there is no reason to expose the USB network itself; instead, a discovery base driver can register the individual devices as they are discovered.

Not all technologies support a discovery mechanism. For example, most serial ports do not support detection, and it is often not even possible to detect whether a device is attached to a serial port.

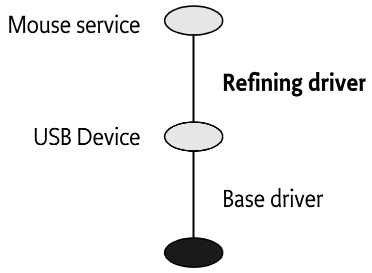
Although each driver bundle should perform discovery on its own, a driver for a non-discoverable serial port requires external help - either through a user interface or by allowing the Configuration Admin service to configure it.

It is possible for the driver bundle to combine automatic discovery of Plug and Play-compliant devices with manual configuration when non-compliant devices are plugged in.

103.4.2.2 Refining Drivers

The second category of device drivers are called *refining drivers*. Refining drivers provide a refined view of a physical device that is already represented by another Device service registered with the Framework. Refining drivers register a Driver service with the Framework. This Driver service is used by the device manager to attach the refining driver to a less refined Device service that is registered as a result of events within the Framework itself.

Figure 103.4 Refining Driver Diagram



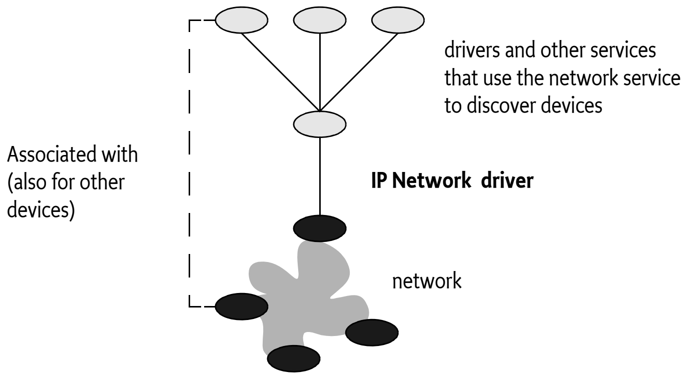
An example of a refining driver is a mouse driver, which is attached to the generic USB Device service representing a physical mouse. It then registers a new Device service which represents it as a Mouse service, defined elsewhere.

The majority of drivers fall into the refining driver type.

103.4.2.3 Network Drivers

An Internet Protocol (IP) capable network such as Ethernet supports individually addressable devices and allows broadcasts, but does not define an intrinsic discovery protocol. In this case, the entire network should be exposed as a single Device service.

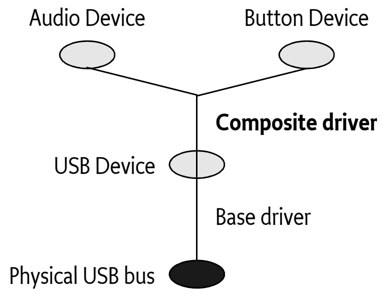
Figure 103.5 Network Driver diagram


drivers and other services that use the network service to discover devices
Associated with (also for other devices)
IP Network driver
network

103.4.2.4 Composite Drivers

Complex devices can often be broken down into several parts. Drivers that attach to a single service and then register multiple Device services are called *composite drivers*. For example, a USB speaker containing software-accessible buttons can be registered by its driver as two separate Device services: an Audio Device service and a Button Device service.

Figure 103.6 Composite Driver structure



This approach can greatly reduce the number of interfaces needed, as well as enhance reusability.

103.4.2.5 Referring Drivers

A referring driver is actually not a driver in the sense that it controls Device services. Instead, it acts as an intermediary to help locate the correct driver bundle. This process is explained in detail in *The Device Attachment Algorithm* on page 55.

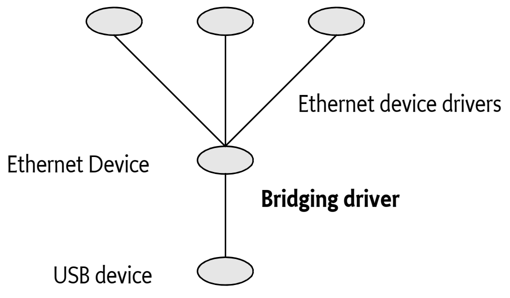
A referring driver implements the call to the `attach` method to inspect the Device service, and decides which Driver bundle would be able to attach to the device. This process can actually involve connecting to the physical device and communicating with it. The `attach` method then returns a `String` object that indicates the `DRIVER_ID` of another driver bundle. This process is called a referral.

For example, a vendor ACME can implement one driver bundle that specializes in recognizing all of the devices the vendor produces. The referring driver bundle does not contain code to control the device - it contains only sufficient logic to recognize the assortment of devices. This referring driver can be small, yet can still identify a large product line. This approach can drastically reduce the amount of downloading and matching needed to find the correct driver bundle.

103.4.2.6 Bridging Drivers

A bridging driver registers a Device service from one device category but attaches it to a Device service from another device category.

Figure 103.7 Bridging Driver Structure



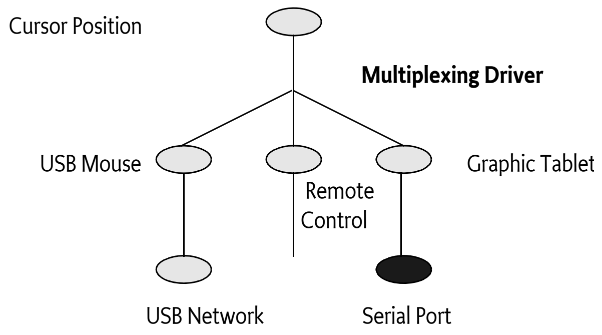
For example, USB to Ethernet bridges exist that allow connection to an Ethernet network through a USB device. In this case, the top level of the USB part of the Device service stack would be an Ethernet Device service. But the same Ethernet Device service can also be the bottom layer of an Ethernet layer of the Device service stack. A few layers up, a bridge could connect into yet another network.

The stacking depth of Device services has no limit, and the same drivers could in fact appear at different levels in the same Device service stack. The graph of drivers-to-Device services roughly mirrors the hardware connections.

103.4.2.7 Multiplexing Drivers

A *multiplexing driver* attaches a number of Device services and aggregates them in a new Device service.

Figure 103.8 Multiplexing Driver Structure

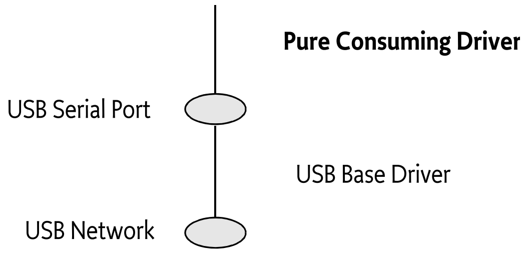


For example, assume that a system has a mouse on USB, a graphic tablet on a serial port, and a remote control facility. Each of these would be registered as a service with the Framework. A multiplexing driver can attach all three, and can merge the different positions in a central Cursor Position service.

103.4.2.8 Pure Consuming Drivers

A *pure consuming driver* bundle will attach to devices without registering a refined version.

Figure 103.9 Pure Consuming Driver Structure



For example, one driver bundle could decide to handle all serial ports through `javax.comm` instead of registering them as services. When a USB serial port is plugged in, one or more Driver services are attached, resulting in a Device service stack with a Serial Port Device service. A pure consuming driver may then attach to the Serial Port Device service and register a new serial port with the `javax.comm.*` registry instead of the Framework service registry. This registration effectively transfers the device from the OSGi environment into another environment.

103.4.2.9 Other Driver Types

It should be noted that any bundle installed in the OSGi environment may get and use a Device service without having to register a Driver service.

The following functionality is offered to those bundles that do register a Driver service and conform to the this specification:

- The bundles can be installed and uninstalled on demand.
- Attachment to the Device service is only initiated after the winning the competition with other drivers.

103.4.3 Driver Service Registration

Drivers are recognized by registering a Driver service with the Framework. This event makes the device manager aware of the existence of the Driver service. A Driver service registration must have a DRIVER_ID property whose value is a String object, uniquely identifying the driver to the device manager. The device manager must use the DRIVER_ID to prevent the installation of duplicate copies of the same driver bundle.

Therefore, this DRIVER_ID must:

- Depend only on the specific behavior of the driver, and thus be independent of unrelated aspects like its location or mechanism of downloading.
- Start with the reversed form of the domain name of the company that implements it: for example, com.acme.widget.1.1.
- Differ from the DRIVER_ID of drivers with different behavior. Thus, it must *also* be different for each revision of the same driver bundle so they may be distinguished.

When a new Driver service is registered, the Device Attachment Algorithm must be applied to each idle Device service. This requirement gives the new Driver service a chance to compete with other Driver services for attaching to idle devices. The techniques outlined in *Optimizations* on page 58 can provide significant shortcuts for this situation.

As a result, the Driver service object can receive match and attach requests before the method which registered the service has returned.

This specification does not define any method for new Driver services to *steal* already attached devices. Once a Device service has been attached by a Driver service, it can only be released by the Driver service itself.

103.4.4 Driver Service Unregistration

When a Driver service is unregistered, it must release all Device services to which it is attached. Thus, *all* its attached Device services become idle. The device manager must gather all of these idle Device services and try to re-attach them. This condition gives other Driver services a chance to take over the refinement of devices after the unregistering driver. The techniques outlined in *Optimizations* on page 58 can provide significant shortcuts for this situation.

A Driver service that is installed by the device manager must remain registered as long as the driver bundle is active. Therefore, a Driver service should only be unregistered if the driver bundle is stopping, an occurrence which may precede its being uninstalled or updated. Driver services should thus not unregister in an attempt to minimize resource consumption. Such optimizations can easily introduce race conditions with the device manager.

103.4.5 Driver Service Methods

The Driver interface consists of the following methods:

- match(ServiceReference) - This method is called by the device manager to find out how well this Driver service matches the Device service as indicated by the ServiceReference argument. The value returned here is specific for a device category. If this Device service is of another device category, the value Device.MATCH_NONE must be returned. Higher values indicate a better match. For the exact matching algorithm, see *The Device Attachment Algorithm* on page 55.

  Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results so that results can be cached by the device manager.

- attach(ServiceReference) - If the device manager decides that a Driver service should be attached to a Device service, it must call this method on the Driver service object. Once this method is called, the Device service is regarded as attached to that Driver service, and no other Driver service must be called to attach to the Device service. The Device service must remain *owned* by the Driver service until the Driver bundle is stopped. No unattach method exists.

  The attach method should return null when the Device service is correctly attached. A referring driver (see *Referring Drivers* on page 48) can return a String object that specifies the DRIVER_ID of a driver that can handle this Device service. In this case, the Device service is not attached and the device manager must attempt to install a Driver service with the same DRIVER_ID via a Driver Locator service. The attach method must be deterministic as described in the previous method.

103.4.6 Idle Driver Bundles

An idle Driver bundle is a bundle with a registered Driver service, and is not attached to any Device service. Idle Driver bundles are consuming resources in the OSGi Framework. The device manager should uninstall bundles that it has installed and which are idle.

103.5 Driver Locator Service

The device manager must automatically install Driver bundles, which are obtained from Driver Locator services, when new Device services are registered.

A Driver Locator service encapsulates the knowledge of how to fetch the Driver bundles needed for a specific Device service. This selection is made on the properties that are registered with a device: for example, DEVICE_CATEGORY and any other properties registered with the Device service registration.

The purpose of the Driver Locator service is to separate the mechanism from the policy. The decision to install a new bundle is made by the device manager (the mechanism), but a Driver Locator service decides which bundle to install and from where the bundle is downloaded (the policy).

Installing bundles has many consequences for the security of the system, and this process is also sensitive to network setup and other configuration details. Using Driver Locator services allows the Operator to choose a strategy that best fits its needs.

Driver services are identified by the DRIVER_ID property. Driver Locator services use this particular ID to identify the bundles that can be installed. Driver ID properties have uniqueness requirements as specified in *Device Service Registration* on page 42. This uniqueness allows the device manager to maintain a list of Driver services and prevent unnecessary installs.

An OSGi Framework can have several different Driver Locator services installed. The device manager must consult all of them and use the combined result set, after pruning duplicates based on the DRIVER_ID values.

103.5.1 The DriverLocator Interface

The DriverLocator interface allows suitable driver bundles to be located, downloaded, and installed on demand, even when completely unknown devices are detected.

It has the following methods:

- findDrivers(Dictionary) - This method returns an array of driver IDs that potentially match a service described by the properties in the Dictionary object. A driver ID is the String object that is registered by a Driver service under the DRIVER_ID property.
- loadDriver(String) - This method returns an InputStream object that can be used to download the bundle containing the Driver service as specified by the driver ID argument. If the Driver Lo-

cator service cannot download such a bundle, it should return null. Once this bundle is downloaded and installed in the Framework, it must register a Driver service with the DRIVER_ID property set to the value of the String argument.

103.5.2 A Driver Example

The following example shows a very minimal Driver service implementation. It consists of two classes. The first class is SerialWidget. This class tracks a single WidgetDevice from *Sample Device Category Specification* on page 44. It registers a javax.comm.SerialPort service, which is a general serial port specification that could also be implemented from other device categories like USB, a COM port, etc. It is created when the SerialWidgetDriver object is requested to attach a WidgetDevice by the device manager. It registers a new javax.comm.SerialPort service in its constructor.

The org.osgi.util.tracker.ServiceTracker is extended to handle the Framework events that are needed to simplify tracking this service. The removedService method of this class is overridden to unregister the SerialPort when the underlying WidgetDevice is unregistered.

```
package com.acme.widget;
import org.osgi.service.device.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;

class SerialWidget extends ServiceTracker
    implements javax.comm.SerialPort,
        org.osgi.service.device.Constants {
    ServiceRegistration registration;

    SerialWidget( BundleContext c, ServiceReference r ) {
        super( c, r, null );
        open();
    }

    public Object addingService( ServiceReference ref ) {
        WidgetDevice dev = (WidgetDevice)
            context.getService( ref );
        registration = context.registerService(
                javax.comm.SerialPort.class.getName(),
                this,
                null        );
            return dev;
    }

    public void removedService( ServiceReference ref,
        Object service ) {
        registration.unregister();
        context.ungetService(ref);
    }
    ... methods for javax.comm.SerialPort that are
    ... converted to underlying WidgetDevice
}
```

A SerialWidgetDriver object is registered with the Framework in the Bundle Activator start method under the Driver interface. The device manager must call the match method for each idle Device service that is registered. If it is chosen by the device manager to control this Device service, a new SerialWidget is created that offers serial port functionality to other bundles.

```
public class SerialWidgetDriver implementsDriver {
    BundleContext context;

    String        spec =
         "(&"
        +" (objectclass=com.acme.widget.WidgetDevice)"
        +" (DEVICE_CATEGORY=WidgetDevice)"
        +" (com.acme.class=Serial)"
        +")";

    Filter        filter;

    SerialWidgetDriver( BundleContext context )
        throws Exception {
        this.context = context;
        filter = context.createFilter(spec);
    }
    public int match( ServiceReference d ) {
        if ( filter.match( d ) )
            return WidgetDevice.MATCH_CLASS;
        else
            return Device.MATCH_NONE;
    }
    public synchronized String attach(ServiceReference r){
        new SerialWidget( context, r );
    }
}
```

103.6 The Driver Selector Service

The purpose of the Driver Selector service is to customize the selection of the best Driver service from a set of suitable Driver bundles. The device manager has a default algorithm as described in *The Device Attachment Algorithm* on page 55. When this algorithm is not sufficient and requires customizing by the operator, a bundle providing a Driver Selector service can be installed in the Framework. This service must be used by the device manager as the final arbiter when selecting the best match for a Device service.

The Driver Selector service is a singleton; only one such service is recognized by the device manager. The Framework method BundleContext.getServiceReference must be used to obtain a Driver Selector service. In the erroneous case that multiple Driver Selector services are registered, the service.ranking property will thus define which service is actually used.

A device manager implementation must invoke the method select(ServiceReference,Match[]). This method receives a Service Reference to the Device service and an array of Match objects. Each Match object contains a link to the ServiceReference object of a Driver service and the result of the match value returned from a previous call to Driver.match. The Driver Selector service should inspect the array of Match objects and use some means to decide which Driver service is best suited. The index of the best match should be returned. If none of the Match objects describe a possible Driver service, the implementation must return DriverSelector.SELECT_NONE (-1).

103.7 Device Manager

Device Access is controlled by the device manager in the background. The device manager is responsible for initiating all actions in response to the registration, modification, and unregistration of Device services and Driver services, using Driver Locator services and a Driver Selector service as helpers.

The device manager detects the registration of Device services and coordinates their attachment with a suitable Driver service. Potential Driver services do not have to be active in the Framework to be eligible. The device manager must use Driver Locator services to find bundles that might be suitable for the detected Device service and that are not currently installed. This selection is done via a DRIVER_ID property that is unique for each Driver service.

The device manager must install and start these bundles with the help of a Driver Locator service. This activity must result in the registration of one or more Driver services. All available Driver services, installed by the device manager and also others, then participate in a bidding process. The Driver service can inspect the Device service through its ServiceReference object to find out how well this Driver service matches the Device service.

If a Driver Selector service is available in the Framework service registry, it is used to decide which of the eligible Driver services is the best match.

If no Driver Selector service is available, the highest bidder must win, with tie breaks defined on the service.ranking and service.id properties. The selected Driver service is then asked to attach the Device service.

If no Driver service is suitable, the Device service remains idle. When new Driver bundles are installed, these idle Device services must be reattached.

The device manager must reattach a Device service if, at a later time, a Driver service is unregistered due to an uninstallation or update. At the same time, however, it should prevent superfluous and non-optimal reattachments. The device manager should also garbage-collect driver bundles it installed which are no longer used.

The device manager is a singleton. Only one device manager may exist, and it must have no public interface.

103.7.1 Device Manager Startup

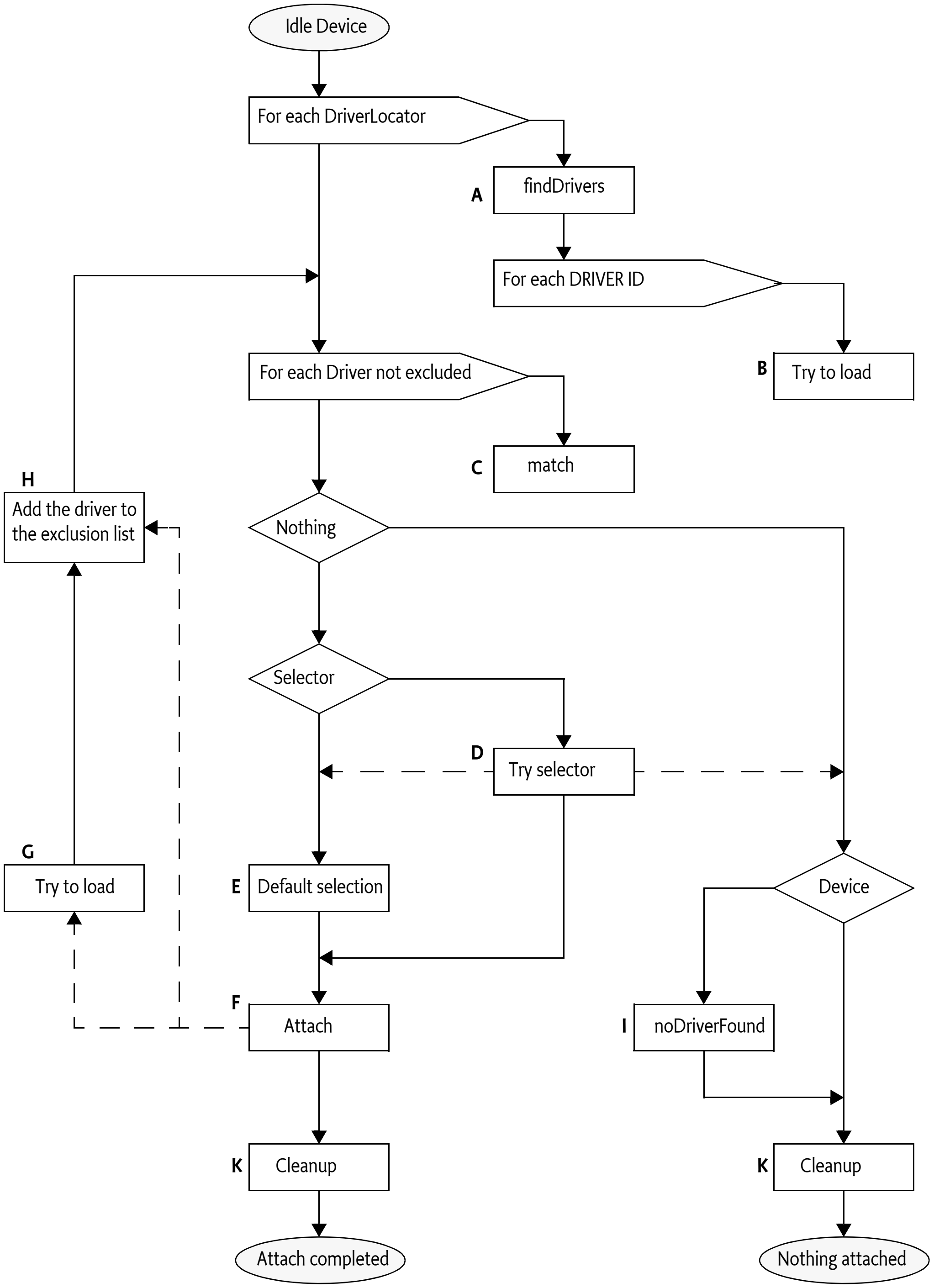
To prevent race conditions during Framework startup, the device manager must monitor the state of Device services and Driver services immediately when it is started. The device manager must not, however, begin attaching Device services until the Framework has been fully started, to prevent superfluous or non-optimal attachments.

The Framework has completed starting when the `FrameworkEvent.STARTED` event has been published. Publication of that event indicates that Framework has finished all its initialization and all bundles are started. If the device manager is started after the Framework has been initialized, it should detect the state of the Framework by examining the state of the system bundle.

103.7.2 The Device Attachment Algorithm

A key responsibility of the device manager is to attach refining drivers to idle devices. The following diagram illustrates the device attachment algorithm.

Figure 103.10 *Device Attachment Algorithm*



103.7.3 Legend

Table 103.3 Driver attachment algorithm

Step	Description
A	DriverLocator.findDrivers is called for each registered Driver Locator service, passing the properties of the newly detected Device service. Each method call returns zero or more DRIVER_ID values (identifiers of particular driver bundles). If the findDrivers method throws an exception, it is ignored, and processing continues with the next Driver Locator service. See *Optimizations* on page 58 for further guidance on handling exceptions.
B	For each found DRIVER_ID that does not correspond to an already registered Driver service, the device manager calls DriverLocator.loadDriver to return an InputStream containing the driver bundle. Each call to loadDriver is directed to one of the Driver Locator services that mentioned the DRIVER_ID in step A. If the loadDriver method fails, the other Driver Locator objects are tried. If they all fail, the driver bundle is ignored. If this method succeeds, the device manager installs and starts the driver bundle. Driver bundles must register their Driver services synchronously during bundle activation.
C	For each Driver service, except those on the exclusion list, call its Driver.match method, passing the ServiceReference object to the Device service. Collect all successful matches - that is, those whose return values are greater than Device.MATCH_NONE - in a list of active matches. A match call that throws an exception is considered unsuccessful and is not added to the list.
D	If there is a Driver Selector service, the device manager calls the DriverSelector.select method, passing the array of active Match objects. If the Driver Selector service returns the index of one of the Match objects from the array, its associated Driver service is selected for attaching the Device service. If the Driver Selector service returns DriverSelector.SELECT_NONE, no Driver service must be considered for attaching the Device service. If the Driver Selector service throws an exception or returns an invalid result, the default selection algorithm is used. Only one Driver Selector service is used, even if there is more than one registered in the Framework. See *The Driver Selector Service* on page 54.
E	The winner is the one with the highest match value. Tie breakers are respectively: • Highest service.ranking property. • Lowest service.id property.
F	The selected Driver service's attach method is called. If the attach method returns null, the Device service has been successfully attached. If the attach method returns a String object, it is interpreted as a referral to another Driver service and processing continues at G. See *Referring Drivers* on page 48. If an exception is thrown, the Driver service has failed, and the algorithm proceeds to try another Driver service after excluding this one from further consideration at Step H.

Step	Description
G	The device manager attempts to load the referred driver bundle in a manner similar to Step B, except that it is unknown which Driver Locator service to use. Therefore, the loadDriver method must be called on each Driver Locator service until one succeeds (or they all fail). If one succeeds, the device manager installs and starts the driver bundle. The driver bundle must register a Driver service during its activation which must be added to the list of Driver services in this algorithm.
H	The referring driver bundle is added to the exclusion list. Because each new referral adds an entry to the exclusion list, which in turn disqualifies another driver from further matching, the algorithm cannot loop indefinitely. This list is maintained for the duration of this algorithm. The next time a new Device service is processed, the exclusion list starts out empty.
I	If no Driver service attached the Device service, the Device service is checked to see whether it implements the Device interface. If so, the noDriverFound method is called. Note that this action may cause the Device service to unregister and possibly a new Device service (or services) to be registered in its place. Each new Device service registration must restart the algorithm from the beginning.
K	Whether an attachment was successful or not, the algorithm may have installed a number of driver bundles. The device manager should remove any idle driver bundles that it installed.

103.7.4 Optimizations

Optimizations are explicitly allowed and even recommended for an implementation of a device manager. Implementations may use the following assumptions:

- Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results.
- The device manager may cache match values and referrals. Therefore, optimizations in the device attachment algorithm based on this assumption are allowed.
- The device manager may delay loading a driver bundle until it is needed. For example, a delay could occur when that DRIVER_ID's match values are cached.
- The results of calls to DriverLocator and DriverSelector methods are not required to be deterministic, and must not be cached by the device manager.
- Thrown exceptions must not be cached. Exceptions are considered transient failures, and the device manager must always retry a method call even if it has thrown an exception on a previous invocation with the same arguments.

103.7.5 Driver Bundle Reclamation

The device manager may remove driver bundles it has installed at any time, provided that all the Driver services in that bundle are idle. This recommended practice prevents unused driver bundles from accumulating over time. Removing driver bundles too soon, however, may cause unnecessary installs and associated delays when driver bundles are needed again.

If a device manager implements driver bundle reclamation, the specified matching algorithm is not guaranteed to terminate unless the device manager takes reclamation into account.

For example, assume that a new Device service triggers the attachment algorithm. A driver bundle recommended by a Driver Locator service is loaded. It does not match, so the Device service remains idle. The device manager is eager to reclaim space, and unloads the driver bundle. The disappearance of the Driver service causes the device manager to reattach idle devices. Because it has not kept a record of its previous activities, it tries to reattach the same device, which closes the loop.

On systems where the device manager implements driver bundle reclamation, all refining drivers should be loaded through Driver Locator services. This recommendation is intended to prevent the device manager from erroneously uninstalling pre-installed driver bundles that cannot later be reinstalled when needed.

The device manager can be updated or restarted. It cannot, however, rely on previously stored information to determine which driver bundles were pre-installed and which were dynamically installed and thus are eligible for removal. The device manager may persistently store cachable information for optimization, but must be able to cold start without any persistent information and still be able to manage an existing connection state, satisfying all of the requirements in this specification.

103.7.6 Handling Driver Bundle Updates

It is not straightforward to determine whether a driver bundle is being updated when the UNREGISTER event for a Driver service is received. In order to facilitate this distinction, the device manager should wait for a period of time after the unregistration for one of the following events to occur:

- A BundleEvent.UNINSTALLED event for the driver bundle.
- A ServiceEvent.REGISTERED event for another Driver service registered by the driver bundle.

If the driver bundle is uninstalled, or if neither of the above events are received within the allotted time period, the driver is assumed to be inactive. The appropriate waiting period is implementation-dependent and will vary for different installations. As a general rule, this period should be long enough to allow a driver to be stopped, updated, and restarted under normal conditions, and short enough not to cause unnecessary delays in reattaching devices. The actual time should be configurable.

103.7.7 Simultaneous Device Service and Driver Service Registration

The device attachment algorithm may discover new driver bundles that were installed outside its direct control, which requires executing the device attachment algorithm recursively. However, in this case, the appearance of the new driver bundles should be queued until completion of the current device attachment algorithm.

Only one device attachment algorithm may be in progress at any moment in time.

The following example sequence illustrates this process when a Driver service is registered:

- Collect the set of all idle devices.
- Apply the device attachment algorithm to each device in the set.
- If no Driver services were registered during the execution of the device attachment algorithm, processing terminates.
- Otherwise, restart this process.

103.8 Security

The device manager is the only privileged bundle in the Device Access specification and requires the org.osgi.framework.AdminPermission with the LIFECYCLE action to install and uninstall driver bundles.

The device manager itself should be free from any knowledge of policies and should not actively set bundle permissions. Rather, if permissions must be set, it is up to the Management Agent to listen to synchronous bundle events and set the appropriate permissions.

Driver Locator services can trigger the download of any bundle, because they deliver the content of a bundle to the privileged device manager and could potentially insert a Trojan horse into the environment. Therefore, Driver Locator bundles need the ServicePermission[DriverLocator, REGISTER] to register Driver Locator services, and the operator should exercise prudence in assigning this ServicePermission.

Bundles with Driver Selector services only require ServicePermission[DriverSelector, REGISTER] to register the DriverSelector service. The Driver Selector service can play a crucial role in the selection of a suitable Driver service, but it has no means to define a specific bundle itself.

103.9 org.osgi.service.device

Device Access Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.device; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.device; version="[1.1,1.2)"
```

103.9.1 Summary

- Constants - This interface defines standard names for property keys associated with Device and Driver services.
- Device - Interface for identifying device services.
- Driver - A Driver service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers.
- DriverLocator - A Driver Locator service can find and load device driver bundles given a property set.
- DriverSelector - When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service.
- Match - Instances of Match are used in the DriverSelector.select(ServiceReference, Match[]) method to identify Driver services matching a Device service.

103.9.2 public interface Constants

This interface defines standard names for property keys associated with Device and Driver services.

The values associated with these keys are of type java.lang.String, unless otherwise stated.

See Also Device, Driver

Since 1.1

No Implement Consumers of this API must not implement this interface

103.9.2.1 public static final String DEVICE_CATEGORY = "DEVICE_CATEGORY"

Property (named "DEVICE_CATEGORY") containing a human readable description of the device categories implemented by a device. This property is of type String[]

Services registered with this property will be treated as devices and discovered by the device manager

103.9.2.2 public static final String DEVICE_DESCRIPTION = "DEVICE_DESCRIPTION"

Property (named "DEVICE_DESCRIPTION") containing a human readable string describing the actual hardware device.

103.9.2.3 public static final String DEVICE_SERIAL = "DEVICE_SERIAL"

Property (named "DEVICE_SERIAL") specifying a device's serial number.

103.9.2.4 public static final String DRIVER_ID = "DRIVER_ID"

Property (named "DRIVER_ID") identifying a driver.

A DRIVER_ID should start with the reversed domain name of the company that implemented the driver (e.g., com.acme), and must meet the following requirements:

- It must be independent of the location from where it is obtained.
- It must be independent of the DriverLocator service that downloaded it.
- It must be unique.
- It must be different for different revisions of the same driver.

This property is mandatory, i.e., every Driver service must be registered with it.

103.9.3 public interface Device

Interface for identifying device services.

A service must implement this interface or use the Constants.DEVICE_CATEGORY registration property to indicate that it is a device. Any services implementing this interface or registered with the DEVICE_CATEGORY property will be discovered by the device manager.

Device services implementing this interface give the device manager the opportunity to indicate to the device that no drivers were found that could (further) refine it. In this case, the device manager calls the noDriverFound() method on the Device object.

Specialized device implementations will extend this interface by adding methods appropriate to their device category to it.

See Also Driver

Concurrency Thread-safe

103.9.3.1 public static final int MATCH_NONE = 0

Return value from Driver.match(ServiceReference) indicating that the driver cannot refine the device presented to it by the device manager. The value is zero.

103.9.3.2 public void noDriverFound()

□ Indicates to this Device object that the device manager has failed to attach any drivers to it.

If this Device object can be configured differently, the driver that registered this Device object may unregister it and register a different Device service instead.

103.9.4 public interface Driver

A Driver service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers. For each newly discovered Device object, the device manager enters a bidding phase. The Driver object whose match(ServiceReference) method bids the highest for a particular Device object will be instructed by the device manager to attach to the Device object.

See Also Device, DriverLocator

Concurrency Thread-safe

103.9.4.1 public String attach(ServiceReference<?> reference) throws Exception

reference the ServiceReference object of the device to attach to

□ Attaches this Driver service to the Device service represented by the given ServiceReference object.

A return value of null indicates that this Driver service has successfully attached to the given Device service. If this Driver service is unable to attach to the given Device service, but knows of a more suitable Driver service, it must return the DRIVER_ID of that Driver service. This allows for the implementation of referring drivers whose only purpose is to refer to other drivers capable of handling a given Device service.

After having attached to the Device service, this driver may register the underlying device as a new service exposing driver-specific functionality.

This method is called by the device manager.

Returns null if this Driver service has successfully attached to the given Device service, or the DRIVER_ID of a more suitable driver

Throws Exception– if the driver cannot attach to the given device and does not know of a more suitable driver

103.9.4.2 public int match(ServiceReference<?> reference) throws Exception

reference the ServiceReference object of the device to match

□ Checks whether this Driver service can be attached to the Device service. The Device service is represented by the given ServiceReference and returns a value indicating how well this driver can support the given Device service, or Device.MATCH_NONE if it cannot support the given Device service at all.

The return value must be one of the possible match values defined in the device category definition for the given Device service, or Device.MATCH_NONE if the category of the Device service is not recognized.

In order to make its decision, this Driver service may examine the properties associated with the given Device service, or may get the referenced service object (representing the actual physical device) to talk to it, as long as it ungets the service and returns the physical device to a normal state before this method returns.

A Driver service must always return the same match code whenever it is presented with the same Device service.

The match function is called by the device manager during the matching process.

Returns value indicating how well this driver can support the given Device service, or Device.MATCH_NONE if it cannot support the Device service at all

Throws Exception– if this Driver service cannot examine the Device service

103.9.5 public interface DriverLocator

A Driver Locator service can find and load device driver bundles given a property set. Each driver is represented by a unique DRIVER_ID.

Driver Locator services provide the mechanism for dynamically downloading new device driver bundles into an OSGi environment. They are supplied by providers and encapsulate all provider-specific details related to the location and acquisition of driver bundles.

See Also Driver

Concurrency Thread-safe

103.9.5.1 public String[] findDrivers(Dictionary<String, ?> props)

props the properties of the device for which a driver is sought

□ Returns an array of DRIVER_ID strings of drivers capable of attaching to a device with the given properties.

The property keys in the specified Dictionary objects are case-insensitive.

Returns array of driver DRIVER_ID strings of drivers capable of attaching to a Device service with the given properties, or null if this Driver Locator service does not know of any such drivers

103.9.5.2 public InputStream loadDriver(String id) throws IOException

id the DRIVER_ID of the driver that needs to be installed.

□ Get an InputStream from which the driver bundle providing a driver with the giving DRIVER_ID can be installed.

Returns An InputStream object from which the driver bundle can be installed or null if the driver with the given ID cannot be located

Throws IOException– the input stream for the bundle cannot be created

103.9.6 public interface DriverSelector

When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service. If at least one Driver service matches, the device manager must choose one. If there is a Driver Selector service registered with the Framework, the device manager will ask it to make the selection. If there is no Driver Selector service, or if it returns an invalid result, or throws an Exception, the device manager uses the default selection strategy.

Since 1.1

Concurrency Thread-safe

103.9.6.1 public static final int SELECT_NONE = -1

Return value from DriverSelector.select, if no Driver service should be attached to the Device service. The value is -1.

103.9.6.2 public int select(ServiceReference<?> reference, Match[] matches)

reference the ServiceReference object of the Device service.

matches the array of all non-zero matches.

□ Select one of the matching Driver services. The device manager calls this method if there is at least one driver bidding for a device. Only Driver services that have responded with nonzero (not Device.MATCH_NONE) match values will be included in the list.

Returns index into the array of Match objects, or SELECT_NONE if no Driver service should be attached

103.9.7 public interface Match

Instances of Match are used in the DriverSelector.select(ServiceReference, Match[]) method to identify Driver services matching a Device service.

See Also DriverSelector

Since 1.1

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

103.9.7.1 public ServiceReference<?> getDriver()

□ Return the reference to a Driver service.

Returns ServiceReference object to a Driver service.

103.9.7.2 public int getMatchValue()

□ Return the match value of this object.

Returns the match value returned by this Driver service.

103.10 References

[1] *Java Communications API*
https://www.oracle.com/java/technologies/java-communications-api.html

[2] *USB Specification*
https://www.usb.org

[3] *Universal Plug and Play*
https://openconnectivity.org/developer/specifications/upnp-resources/upnp/

[4] *Jini, Service Discovery and Usage*
https://en.wikipedia.org/wiki/Jini

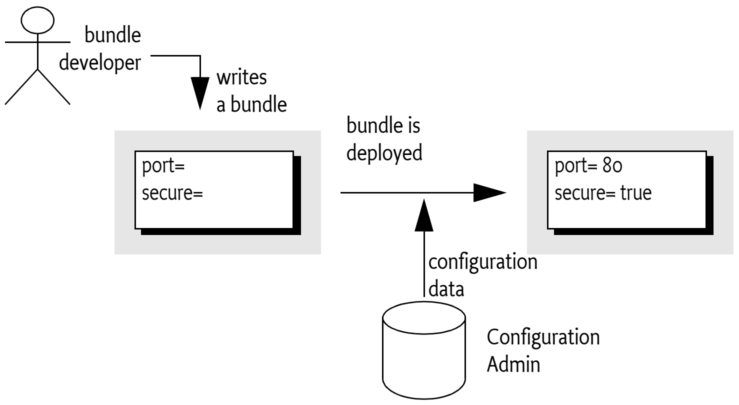
104 Configuration Admin Service Specification

Version 1.6

104.1 Introduction

The Configuration Admin service is an important aspect of the deployment of an OSGi framework. It allows an Operator to configure deployed bundles. Configuring is the process of defining the configuration data for bundles and assuring that those bundles receive that data when they are active in the OSGi framework.

Figure 104.1 Configuration Admin Service Overview



104.1.1 Essentials

The following requirements and patterns are associated with the Configuration Admin service specification:

- *Local Configuration* - The Configuration Admin service must support bundles that have their own user interface to change their configurations.
- *Reflection* - The Configuration Admin service must be able to deduce the names and types of the needed configuration data.
- *Legacy* - The Configuration Admin service must support configuration data of existing entities (such as devices).
- *Object Oriented* - The Configuration Admin service must support the creation and deletion of instances of configuration information so that a bundle can create the appropriate number of services under the control of the Configuration Admin service.
- *Embedded Devices* - The Configuration Admin service must be deployable on a wide range of platforms. This requirement means that the interface should not assume file storage on the platform. The choice to use file storage should be left to the implementation of the Configuration Admin service.

- *Remote versus Local Management* - The Configuration Admin service must allow for a remotely managed OSGi framework, and must not assume that con-figuration information is stored locally. Nor should it assume that the Configuration Admin service is always done remotely. Both implementation approaches should be viable.
- *Availability* - The OSGi environment is a dynamic environment that must run continuously (24/7/365). Configuration updates must happen dynamically and should not require restarting of the system or bundles.
- *Immediate Response* - Changes in configuration should be reflected immediately.
- *Execution Environment* - The Configuration Admin service will not require more than an environment that fulfills the minimal execution requirements.
- *Communications* - The Configuration Admin service should not assume "always-on" connectivity, so the API is also applicable for mobile applications in cars, phones, or boats.
- *Extendability* - The Configuration Admin service should expose the process of configuration to other bundles. This exposure should at a minimum encompass initiating an update, removing certain configuration properties, adding properties, and modifying the value of properties potentially based on existing property or service values.
- *Complexity Trade-offs* - Bundles in need of configuration data should have a simple way of obtaining it. Most bundles have this need and the code to accept this data. Additionally, updates should be simple from the perspective of the receiver.

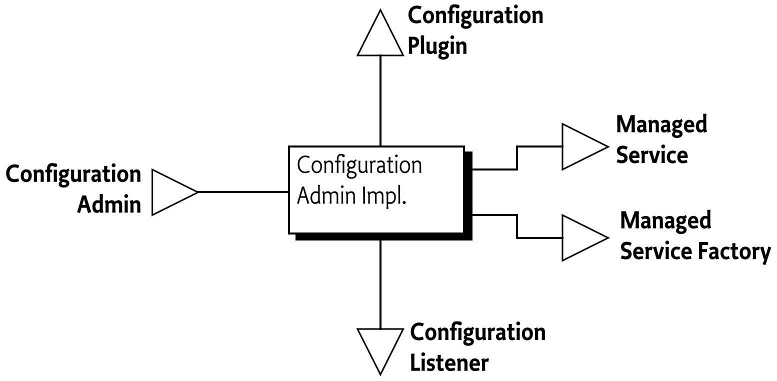
  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.
- *Regions* - It should be possible to create groups of bundles and a manager in a single system that share configuration data that is not accessible outside the region.
- *Shared Information* - It should be possible to share configuration data between bundles.

104.1.2 Entities

- *Configuration information* - The information needed by a bundle before it can provide its intended functionality.
- *Configuration dictionary* - The configuration information when it is passed to the target service. It consists of a `Dictionary` object with a number of properties and identifiers.
- *Configuring Bundle* - A bundle that modifies the configuration information through the Configuration Admin service. This bundle is either a management bundle or the bundle for which the configuration information is intended.
- *Configuration Target* - The target service that will receive the configuration information. For services, there are two types of targets: `ManagedServiceFactory` or `ManagedService` objects.
- *Configuration Admin Service* - This service is responsible for supplying configuration target bundles with their configuration information. It maintains a database with configuration information, keyed on the `service.pid` of configuration target services. These services receive their configuration dictionary/dictionaries when they are registered with the Framework. Configurations can be modified or extended using Configuration Plugin services before they reach the target bundle.
- *Managed Service* - A Managed Service represents a client of the Configuration Admin service, and is thus a configuration target. Bundles should register a Managed Service to receive the configuration data from the Configuration Admin service. A Managed Service adds one or more unique `service.pid` service properties as a primary key for the configuration information.
- *Managed Service Factory* - A Managed Service Factory can receive a number of configuration dictionaries from the Configuration Admin service, and is thus also a configuration target service. It should register with one or more `service.pid` strings and receives zero or more configuration dictionaries. Each dictionary has its own PID that is distinct from the factory PID.

- *Configuration Object* - Implements the Configuration interface and contains the configuration dictionary for a Managed Service or one of the configuration dictionaries for a Managed Service Factory. These objects are manipulated by configuring bundles.
- *Configuration Plugin Services* - Configuration Plugin services are called before the configuration dictionary is given to the configuration targets. The plug-in can modify the configuration dictionary, which is passed to the Configuration Target.

Figure 104.2 Overall Service Diagram



104.1.3 Synopsis

This specification is based on the concept of a Configuration Admin service that manages the configuration of an OSGi framework. It maintains a database of Configuration objects, locally or remotely. This service monitors the service registry and provides configuration information to services that are registered with a service.pid property, the Persistent IDentity (PID), and implement one of the following interfaces:

- *Managed Service* - A service registered with this interface receives its *configuration dictionary* from the database or receives null when no such configuration exists.
- *Managed Service Factory* - Services registered with this interface can receive several configuration dictionaries when registered. The database contains zero or more configuration dictionaries for this service. Each configuration dictionary is given sequentially to the service.

The database can be manipulated either by the Management Agent or bundles that configure themselves. Other parties can provide Configuration Plugin services. Such services participate in the configuration process. They can inspect the configuration dictionary and modify it before it reaches the target service.

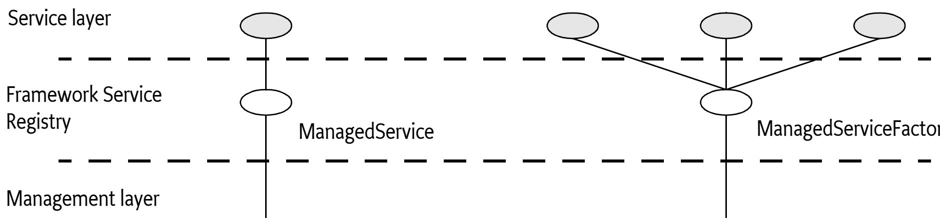
104.2 Configuration Targets

One of the more complicated aspects of this specification is the subtle distinction between the ManagedService and ManagedServiceFactory classes. Both receive configuration information from the Configuration Admin service and are treated similarly in most respects. Therefore, this specification refers to *configuration targets* or simply *targets* when the distinction is irrelevant.

The difference between these types is related to the cardinality of the configuration dictionary. A Managed Service is used when an existing entity needs a configuration dictionary. Thus, a one-to-one relationship always exists between the configuration dictionary and the configurable entity in the Managed Service. There can be multiple Managed Service targets registered with the same PID but a Managed Service can only configure a single entity in each given Managed Service.

A Managed Service Factory is used when part of the configuration is to define *how many instances are required* for a given Managed Service Factory. A management bundle can create, modify, and delete any number of instances for a Managed Service Factory through the Configuration Admin service. Each instance is configured by a single `Configuration` object. Therefore, a Managed Service Factory can have multiple associated `Configuration` objects.

Figure 104.3 Differentiation of ManagedService and ManagedServiceFactory Classes



A Configuration target updates the target when the underlying Configuration object is created, updated, or deleted. However, it is not called back when the Configuration Admin service is shutdown or the service is ungotten.

To summarize:

- A *Managed Service* must receive a single configuration dictionary when it is registered or when its configuration is modified.
- A *Managed Service Factory* must receive from zero to *n* configuration dictionaries when it registers, depending on the current configuration. The Managed Service Factory is informed of configuration dictionary changes: modifications, creations, and deletions.

104.3 The Persistent Identity

A crucial concept in the Configuration Admin service specification is the Persistent IDentity (PID) as defined in the Framework's service layer. Its purpose is to act as a primary key for objects that need a configuration dictionary. The name of the service property for PID is defined in the Framework in `org.osgi.framework.Constants.SERVICE_PID`.

The Configuration Admin service requires the use of one or more PIDs with Managed Service and Managed Service Factory registrations because it associates its configuration data with PIDs.

A service can register with multiple PIDs and PIDs can be shared between multiple targets (both Managed Service and Managed Service Factory targets) to receive the same information. If PIDs are to be shared between Bundles then the location of the Configuration must be a multi-location, see *Location Binding* on page 71.

The Configuration Admin must track the configuration targets on their actual PID. That is, if the `service.pid` service property is modified then the Configuration Admin must treat it as if the service was unregistered and then re-registered with the new PID.

104.3.1 PID Syntax

PIDs are intended for use by other bundles, not by people, but sometimes the user is confronted with a PID. For example, when installing an alarm system, the user needs to identify the different components to a wiring application. This type of application exposes the PID to end users.

PIDs should follow the symbolic-name syntax, which uses a very restricted character set. The following sections define some schemes for common cases. These schemes are not required, but bundle developers are urged to use them to achieve consistency.

104.3.1.1 Local Bundle PIDs

As a convention, descriptions starting with the bundle identity and a full stop ('.' \u002E) are reserved for a bundle. As an example, a PID of "65.536" would belong to the bundle with a bundle identity of 65.

104.3.1.2 Software PIDs

Configuration target services that are singletons can use a Java package name they own as the PID (the reverse domain name scheme) as long as they do not use characters outside the basic ASCII set. As an example, the PID named com.acme.watchdog would represent a Watchdog service from the ACME company.

104.3.1.3 Devices

Devices are usually organized on buses or networks. The identity of a device, such as a unique serial number or an address, is a good component of a PID. The format of the serial number should be the same as that printed on the housing or box, to aid in recognition.

Table 104.1 Schemes for Device-Oriented PID Names

Bus	Example	Format	Description
USB	USB.0123-0002-9909873	idVendor (hex 4) idProduct (hex 4) iSerialNumber (decimal)	Universal Serial Bus. Use the standard device descriptor.
IP	IP.172.16.28.21	IP nr (dotted decimal)	Internet Protocol
802	802-00:60:97:00:9A:56	MAC address with : separators	IEEE 802 MAC address (Token Ring, Ethernet,...)
ONE	ONE.06-00000021E461	Family (hex 2) and serial number including CRC (hex 6)	1-wire bus of Dallas Semiconductor
COM	COM.krups-brewer-12323	serial number or type name of device	Serial ports

104.3.2 Targeted PIDs

PIDs are defined as primary keys for the configuration object; any target that uses the PID in its service registration (and has the proper permissions if security is on) will receive the configuration associated with it, regardless of the bundle that registered the target service. Though in general the PID is designed to ignore the bundle, there are a number of cases where the bundle becomes relevant. The most typical case is where a bundle is available in different versions. Each version will request the same PID and will get therefore configured identically.

Targeted PIDs are specially formatted PIDs that are interpreted by the Configuration Admin service. Targeted PIDs work both as a normal Managed Service PID and as a Managed Service Factory PID. In the case of factories, the targeted PID is the Factory PID since the other PID is chosen by CM for each instance.

The target PID scopes the applicability of the PID to a limited set of target bundles. The syntax of a target pid is:

```
target-pid  ::=  PID
    ( '|' symbolic-name ( '|' version ( '|' location )? )? )?
```

Targets never register with a target PID, target PIDs should only be used when creating, getting, or deleting a Configuration through the Configuration Admin service. The target PID is still the primary key of the Configuration and is thus in itself a PID. The distinction is only made when the Configuration Admin must update a target service. Instead of using the non-target PID as the primary key it must first search if there exists a target PID in the Configuration store that matches the requested target PID.

When a target registers and needs to be updated the Configuration Admin must first find the Configuration with the *best matching* PID. It must logically take the requested PID, append it with the bundle symbolic name, the bundle version, and the bundle location. The version must be formatted canonically, that is, according to the toString() method of the Version class. The rules for best matching are then as follows:

Look for a Configuration, in the given order, with a key of:

```
<pid>|<bsn>|<version>|<location>
<pid>|<bsn>|<version>
<pid>|<bsn>
<pid>
```

For example:

```
com.example.web.WebConf|com.acme.example|3.2.0|http://www.xyz.com/acme.jar
com.example.web.WebConf|com.acme.example|3.2.0
com.example.web.WebConf|com.acme.example
com.example.web.WebConf
```

If a registered target service has a PID that contains a vertical line ('|' \u007c)| then the value must be taken as is and must not be interpreted as a targeted PID.

The service.pid configuration property for a targeted PID configuration must always be set to the targeted PID. That is, if the PID is com.example.web.WebConf and the targeted PID com.example.web.WebConf|com.acme.example|3.2.0 then the property in the Configuration dictionary must be the targeted PID.

If a Configuration with a targeted PID is deleted or a Configuration with a new targeted PID is added then all targets that would be stale must be reevaluated against the new situation and updated accordingly if they are no longer bound against the best matching target PID.

104.3.3 Extenders and Targeted PIDs

Extenders like Declarative Services use Configurations but bypass the general Managed Service or Managed Service Factory method. It is the responsibility of these extenders to access the Configurations using the targeted PIDs.

Since getting a Configuration tends to create that Configuration it is necessary for these extenders to use the listConfigurations(String) method to find out if a more targeted Configuration exists. There are many ways the extender can find the most targeted PID. For example, the following code gets the most targeted PID for a given bundle.

```
String mostTargeted(String key, String pid, Bundle bundle) throws Exception {
    String bsn = bundle.getSymbolicName();
    Version version = bundle.getVersion();
    String location = bundle.getLocation();
    String f = String.format("(|(%1$s=%2$s)(%1$s=%2$s|%3$s)" +
        "(%1$s=%2$s|%3$s|%4$s)(%1$s=%2$s|%3$s|%4$s|%5$s))",
        key, pid, bsn, version, location );

    Configuration[] configurations = cm.listConfigurations(f);
    if (configurations == null)
        return null;

    String largest = null;
    for (Configuration c : configurations) {
        String s = (String) c.getProperties().get(key);
        if ((largest == null) || (largest.length() < s.length()))
            largest = s;
    }
    return largest;
}
```

104.4 The Configuration Object

A Configuration object contains the configuration dictionary, which is a set of properties that configure an aspect of a bundle. A bundle can receive Configuration objects by registering a configuration target service with a PID service property. See *The Persistent Identity* on page 68 for more information about PIDs.

During registration, the Configuration Admin service must detect these configuration target services and hand over their configuration dictionary via a callback. If this configuration dictionary is subsequently modified, the modified dictionary is handed over to the configuration target with the same callback.

The Configuration object is primarily a set of properties that can be updated by a Management Agent, user interfaces on the OSGi framework, or other applications. Configuration changes are first made persistent, and then passed to the target service via a call to the updated method in the ManagedServiceFactory or ManagedService class.

A Configuration object must be uniquely bound to a Managed Service or Managed Service Factory. This implies that a bundle must not register a Managed Service Factory with a PID that is the same as the PID given to a Managed Service.

104.4.1 Location Binding

When a Configuration object is created with either getConfiguration(String), getFactoryConfiguration(String,String), or createFactoryConfiguration(String), it becomes *bound* to the location of the calling bundle. This location is obtained with the getBundleLocation() method.

Location binding is a security feature that assures that only management bundles can modify configuration data, and other bundles can only modify their own configuration data. A Security Exception is thrown if a bundle does not have ConfigurationPermission[location, CONFIGURE].

The two argument versions of getConfiguration(String,String) and createFactoryConfiguration(String,String) as well as the three argument version of getFactoryConfiguration(String,String,String) take a location String as their last argument. These methods require the correct permission, and they create Configuration objects bound to the specified location.

Locations can be specified for a specific Bundle or use *multi-locations*. For a specific location the Configuration location must exactly match the location of the target's Bundle. A multi-location is any location that has the following syntax:

```
multi-location ::= '?' symbolic-name?
```

For example

```
?com.acme
```

The path after the question mark is the *multi-location name*, the multi-location name can be empty if only a question mark is specified. Configurations with a multi-location are dispatched to any target that has *visibility* to the Configuration. The visibility for a given Configuration c depends on the following rules:

- *Single-Location* - If c.location is not a multi-location then a Bundle only has visibility if the Bundle's location exactly matches c.location. In this case there is never a security check.
- *Multi-Location* - If c.location is a multi-location (that is, starts with a question mark):
  - *Security Off* - The Bundle always has visibility
  - *Security On* - The target's Bundle must have ConfigurationPermission[c.location, TARGET] as defined by the Bundle's hasPermission method. The resource name of the permission must include the question mark.

The permission matches on the whole name, including any leading ?. The TARGET action is only applicable in the multi-location scenario since the security is not checked for a single-location. There is therefore no point in granting a Bundle a permission with TARGET action for anything but a multi-location (starting with a ?).

It is therefore possible to register services with the same PID from different bundles. If a multi-location is used then each bundle will be evaluated for a corresponding configuration update. If the bundle has visibility then it is updated, otherwise it is not.

If multiple targets must be updated then the order of updating is the ranking order of their services.

If a target loses visibility because the Configuration's location changes then it must immediately be deleted from the perspective of that target. That is, the target must see a deletion (Managed Service Factory) or an update with null (Managed Service). If a configuration target gains visibility then the target must see a new update with the proper configuration dictionary. However, the associated events must not be sent as the underlying Configuration is not actually deleted nor modified.

Changes in the permissions must not initiate a recalculation of the visibility. If the permissions are changed this will not become visible until one of the other events happen that cause a recalculation of the visibility.

If the location is changed then the Configuration Admin must send a CM_LOCATION_CHANGED event to signal that the location has changed. It is up to the Configuration Listeners to update their state appropriately.

104.4.2 Dynamic Binding

Dynamic binding is available for backward compatibility with earlier versions. It is recommended that management agents explicitly set the location to a ? (a multi-location) to allow multiple bundles to share PIDs and not use the dynamic binding facility. If a management agent uses ?, it must at least have ConfigurationPermission[?, CONFIGURE] when security is on, it is also possible to use ConfigurationPermission[?*, CONFIGURE] to not limit the management agent. See *Regions* on page 84 for some examples of using the locations in isolation scenarios.

A null location parameter can be used to create Configuration objects that are not yet bound. In this case, the Configuration becomes bound to a specific location the first time that it is compared to a Bundle's location. If a bundle becomes dynamically bound to a Configuration then a CM_LOCATION_CHANGED event must be dispatched.

When this *dynamically bound* Bundle is subsequently uninstalled, configurations that are bound to this bundle must be released. That means that for such Configuration object's the bundle location must be set to null again so it can be bound again to another bundle.

104.4.3 Configuration Properties

A configuration dictionary contains a set of properties in a Dictionary object. The value of the property must be the same type as the set of Primary Property Types specified in *OSGi Core Release 8* Filter Syntax.

The name or key of a property must always be a String object, and is not case-sensitive during look up, but must preserve the original case. The format of a property name should be:

```
property-name ::= public | private
public        ::= symbolic-name // See General Syntax in Core Framework
private       ::= '.' symbolic-name
```

Properties can be used in other subsystems that have restrictions on the character set that can be used. The symbolic-name production uses a very minimal character set.

Bundles must not use nested lists or arrays, nor must they use mixed types. Using mixed types or nesting makes it impossible to use the meta typing specification. See *Metatype Service Specification* on page 115.

Property values that are collections may have an ordering that must be preserved when persisting the configuration so that later access to the property value will see the preserved ordering of the collection.

104.4.4 Property Propagation

A configuration target should copy the public configuration properties (properties whose name does not start with a '.' or \u002E) of the Dictionary object argument in updated(Dictionary) into the service properties on any resulting service registration.

This propagation allows the development of applications that leverage the Framework service registry more extensively, so compliance with this mechanism is advised.

A configuration target may ignore any configuration properties it does not recognize, or it may change the values of the configuration properties before these properties are registered as service properties. Configuration properties in the Framework service registry are not strictly related to the configuration information.

Bundles that follow this recommendation to propagate public configuration properties can participate in horizontal applications. For example, an application that maintains physical location information in the Framework service registry could find out where a particular device is located in the house or car. This service could use a property dedicated to the physical location and provide functions that leverage this property, such as a graphic user interface that displays these locations.

Bundles performing service registrations on behalf of other bundles (e.g. OSGi Declarative Services) should propagate all public configuration properties and not propagate private configuration properties.

104.4.5 Automatic Properties

The Configuration Admin service must automatically add a number of properties to the configuration dictionary. If these properties are also set by a configuring bundle or a plug-in, they must always be overridden before they are given to the target service, see *Configuration Plugin* on page 87. Therefore, the receiving bundle or plug-in can assume that the following properties are defined by the Configuration Admin service and not by the configuring bundle:

- service.pid - Set to the PID of the associated Configuration object. This is the full the targeted PID if a targeted PID is used, see *Targeted PIDs* on page 69.
- service.factoryPid - Only set for a Managed Service Factory. It is then set to the PID of the associated Managed Service Factory. This is the full the targeted PID if a targeted PID is used.
- service.bundleLocation - Set to the location of the Configuration object. This property can only be used for searching, it may not appear in the configuration dictionary returned from the getProperties method due to security reasons, nor may it be used when the target is updated.

Constants for some of these properties can be found in org.osgi.framework.Constants and the ConfigurationAdmin interface. These service properties are all of type String.

104.4.6 Equality

Two different Configuration objects can actually represent the same underlying configuration. This means that a Configuration object must implement the equals and hashCode methods in such a way that two Configuration objects are equal when their PID is equal.

104.5 Managed Service

A Managed Service is used by a bundle that needs one or more configuration dictionaries. It therefore registers the Managed Service with one or more PIDs and is thus associated with one Configuration object in the Configuration Admin service for each registered PID. A bundle can register any number of ManagedService objects, but each must be identified with its own PID or PIDs.

A bundle should use a Managed Service when it needs configuration information for the following:

- *A Singleton* - A single entity in the bundle that needs to be configured.
- *Externally Detected Devices* - Each device that is detected causes a registration of an associated ManagedService object. The PID of this object is related to the identity of the device, such as the address or serial number.

A Managed Service may be registered with more than one PID and therefore be associated with multiple Configuration objects, one for each PID. Using multiple PIDs for a Managed Service is not recommended. For example, when a configuration is deleted for a Managed Service there is no way to identify which PID is associated with the deleted configuration.

104.5.1 Singletons

When an object must be instantiated only once, it is called a *singleton*. A singleton requires a single configuration dictionary. Bundles may implement several different types of singletons if necessary.

For example, a Watchdog service could watch the registry for the status and presence of services in the Framework service registry. Only one instance of a Watchdog service is needed, so only a single configuration dictionary is required that contains the polling time and the list of services to watch.

104.5.2 Networks

When a device in the external world needs to be represented in the OSGi Environment, it must be detected in some manner. The Configuration Admin service cannot know the identity and the number of instances of the device without assistance. When a device is detected, it still needs configuration information in order to play a useful role.

For example, a 1-Wire network can automatically detect devices that are attached and removed. When it detects a temperature sensor, it could register a Sensor service with the Framework service registry. This Sensor service needs configuration information specifically for that sensor, such as which lamps should be turned on, at what temperature the sensor is triggered, what timer should be started, in what zone it resides, and so on. One bundle could potentially have hundreds of these sensors and actuators, and each needs its own configuration information.

Each of these Sensor services should be registered as a Managed Service with a PID related to the physical sensor (such as the address) to receive configuration information.

Other examples are services discovered on networks with protocols like Jini, UPnP, and Salutation. They can usually be represented in the Framework service registry. A network printer, for example, could be detected via UPnP. Once in the service registry, these services usually require local configuration information. A Printer service needs to be configured for its local role: location, access list, and so on.

This information needs to be available in the Framework service registry whenever that particular Printer service is registered. Therefore, the Configuration Admin service must remember the configuration information for this Printer service.

This type of service should register with the Framework as a Managed Service in order to receive appropriate configuration information.

104.5.3 Configuring Managed Services

A bundle that needs configuration information should register one or more `ManagedService` objects with a PID service property. If it has a default set of properties for its configuration, it may include them as service properties of the Managed Service. These properties may be used as a configuration template when a `Configuration` object is created for the first time. A Managed Service optionally implements the `MetaTypeProvider` interface to provide information about the property types. See *Meta Typing* on page 89.

When this registration is detected by the Configuration Admin service, the following steps must occur:

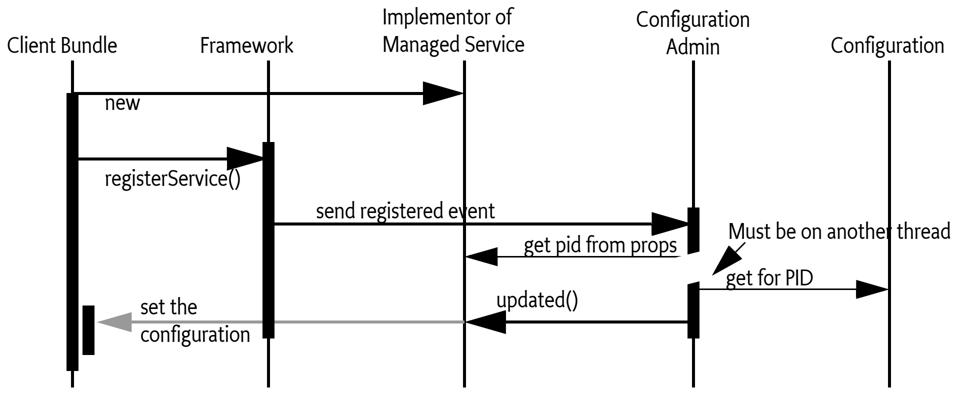
- The configuration stored for the registered PID must be retrieved. If there is a `Configuration` object for this PID and the configuration is visible for the associated bundle then it is sent to the Managed Service with `updated(Dictionary)`.
- If a Managed Service is registered and no configuration information is available or the configuration is not visible then the Configuration Admin service must call `updated(Dictionary)` with a `null` parameter.
- If the Configuration Admin service starts *after* a Managed Service is registered, it must call `updated(Dictionary)` on this service as soon as possible according to the prior rules. For this reason, a Managed Service must always get a callback when it registers *and* the Configuration Admin service is started.

Multiple Managed Services can register with the same PID, they are all updated as long as they have visibility to the configuration as defined by the location, see *Location Binding* on page 71.

If the Managed Service is registered with more than one PID and more than one PID has no configuration information available, then `updated(Dictionary)` will be called multiple times with a `null` parameter.

The `updated(Dictionary)` callback from the Configuration Admin service to the Managed Service must take place asynchronously. This requirement allows the Managed Service to finish its initialization in a synchronized method without interference from the Configuration Admin service callback. Care should be taken not to cause deadlocks by calling the Framework within a synchronized method.

Figure 104.4 Managed Service Configuration Action Diagram



The updated method may throw a ConfigurationException. This object must describe the problem and what property caused the exception.

104.5.4 Race Conditions

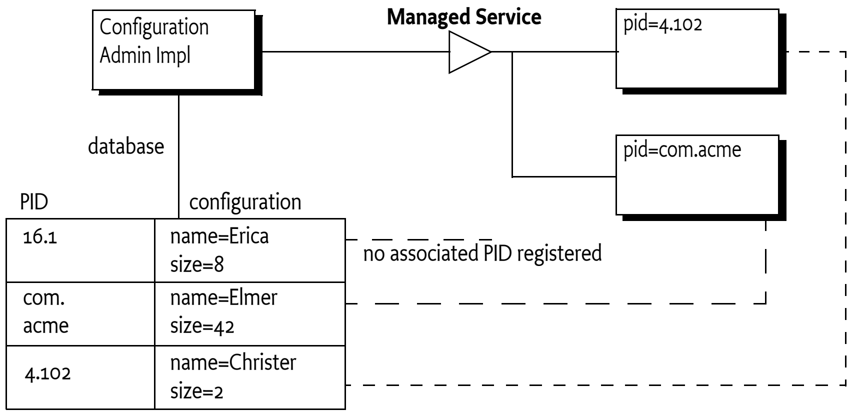
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

In cases where race conditions could be harmful, the Managed Service must be split into two pieces: an object performing the actual service and a Managed Service. First, the Managed Service is registered, the configuration is received, and the actual service object is registered. In such cases, the use of a Managed Service Factory that performs this function should be considered.

104.5.5 Examples of Managed Service

Figure 104.5 shows a Managed Service configuration example. Two services are registered under the ManagedService interface, each with a different PID.

Figure 104.5 PIDs and External Associations



The Configuration Admin service has a database containing a configuration record for each PID. When the Managed Service with service.pid = com.acme is registered, the Configuration Admin service will retrieve the properties name=Elmer and size=42 from its database. The properties are stored in a Dictionary object and then given to the Managed Service with the updated(Dictionary) method.

104.5.5.1 Configuring A Console Bundle

In this example, a bundle can run a single debugging console over a Telnet connection. It is a singleton, so it uses a ManagedService object to get its configuration information: the port and the network name on which it should register.

```
class SampleManagedService implements ManagedService{
    Dictionary          properties;
    ServiceRegistration registration;
    Console             console;

    public void start(
        BundleContext context ) throws Exception {
        properties = new Hashtable();
        properties.put( Constants.SERVICE_PID,
            "com.acme.console" );

        registration = context.registerService(
            ManagedService.class.getName(),
            this,
            properties
        );
    }

    public synchronized void updated( Dictionary np ) {
        if ( np != null ) {
            properties = np;
            properties.put(
                Constants.SERVICE_PID, "com.acme.console" );
        }

        if (console == null)
            console = new Console();

        int port = ((Integer)properties.get("port"))
            .intValue();

        String network = (String) properties.get("network");
        console.setPort(port, network);
        registration.setProperties(properties);
    }
    ... further methods
}
```

104.5.6 Deletion

When a Configuration object for a Managed Service is deleted, the Configuration Admin service must call updated(Dictionary) with a null argument on a thread that is different from that on which the Configuration.delete was executed. This deletion must send out a Configuration Event CM_DELETED asynchronously to any registered Configuration Listener services after the updated method is called with a null.

104.6 Managed Service Factory

A Managed Service Factory is used when configuration information is needed for a service that can be instantiated multiple times. When a Managed Service Factory is registered with the Framework, the Configuration Admin service consults its database and calls updated(String,Dictionary) for each associated and visible Configuration object that matches the PIDs on the registration. It passes the identifier of the Configuration instance, which can be used as a PID, as well as a Dictionary object with the configuration properties.

A Managed Service Factory is useful when the bundle can provide functionality a number of times, each time with different configuration dictionaries. In this situation, the Managed Service Factory acts like a *class* and the Configuration Admin service can use this Managed Service Factory to *instantiate instances* for that *class*.

In the next section, the word *factory* refers to this concept of creating *instances* of a function defined by a bundle that registers a Managed Service Factory.

104.6.1 When to Use a Managed Service Factory

A Managed Service Factory should be used when a bundle does not have an internal or external entity associated with the configuration information but can potentially be instantiated multiple times.

104.6.1.1 Example Email Fetcher

An email fetcher program displays the number of emails that a user has - a function likely to be required for different users. This function could be viewed as a *class* that needs to be *instantiated* for each user. Each instance requires different parameters, including password, host, protocol, user id, and so on.

An implementation of the Email Fetcher service should register a `ManagedServiceFactory` object. In this way, the Configuration Admin service can define the configuration information for each user separately. The Email Fetcher service will only receive a configuration dictionary for each required instance (user).

104.6.1.2 Example Temperature Conversion Service

Assume a bundle has the code to implement a conversion service that receives a temperature and, depending on settings, can turn an actuator on and off. This service would need to be instantiated many times depending on where it is needed. Each instance would require its own configuration information for the following:

- Upper value
- Lower value
- Switch Identification
- ...

Such a conversion service should register a service object under a `ManagedServiceFactory` interface. A configuration program can then use this Managed Service Factory to create instances as needed. For example, this program could use a Graphic User Interface (GUI) to create such a component and configure it.

104.6.1.3 Serial Ports

Serial ports cannot always be used by the OSGi Device Access specification implementations. Some environments have no means to identify available serial ports, and a device on a serial port cannot always provide information about its type.

Therefore, each serial port requires a description of the device that is connected. The bundle managing the serial ports would need to instantiate a number of serial ports under the control of the Configuration Admin service, with the appropriate `DEVICE_CATEGORY` property to allow it to participate in the Device Access implementation.

If the bundle cannot detect the available serial ports automatically, it should register a Managed Service Factory. The Configuration Admin service can then, with the help of a configuration program, define configuration information for each available serial port.

104.6.2 Registration

Similar to the Managed Service configuration dictionary, the configuration dictionary for a Managed Service Factory is identified by a PID. The Managed Service Factory, however, also has a *factory* PID, which is the PID of the associated Managed Service Factory. It is used to group all Managed Service Factory configuration dictionaries together.

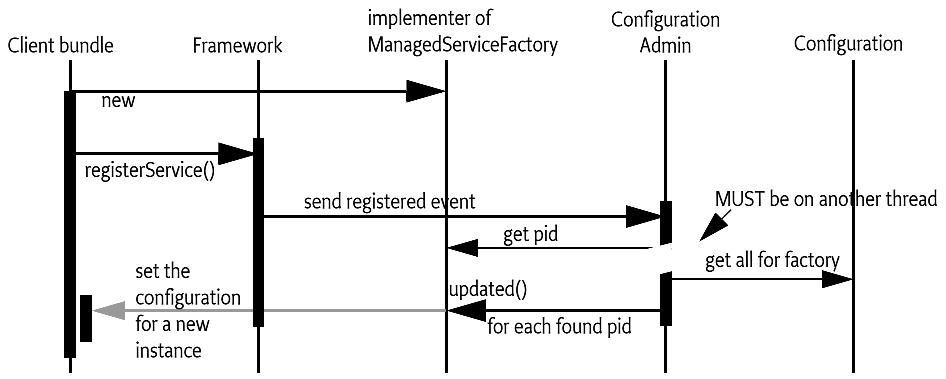
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all visible configuration dictionaries for this factory and must then sequentially call `ManagedServiceFactory.updated(String,Dictionary)` for each configuration dictionary. The first argument is the PID of the `Configuration` object (the one created by the Configuration Admin service) and the second argument contains the configuration properties.

The Managed Service Factory should then create any artifacts associated with that factory. Using the PID given in the Configuration object, the bundle may register new services (other than a Managed Service) with the Framework, but this is not required. This may be necessary when the PID is useful in contexts other than the Configuration Admin service.

The receiver must *not* register a Managed Service with this PID because this would force two Configuration objects to have the same PID. If a bundle attempts to do this, the Configuration Admin service should log an error and must ignore the registration of the Managed Service.

The Configuration Admin service must guarantee that no race conditions exist between initialization, updates, and deletions.

Figure 104.6 Managed Service Factory Action Diagram



A Managed Service Factory has only one update method: updated(String,Dictionary). This method can be called any number of times as Configuration objects are created or updated.

The Managed Service Factory must detect whether a PID is being used for the first time, in which case it should create a new *instance*, or a subsequent time, in which case it should update an existing instance.

The Configuration Admin service must call updated(String,Dictionary) on a thread that is different from the one that executed the registration. This requirement allows an implementation of a Managed Service Factory to use a synchronized method to assure that the callbacks do not interfere with the Managed Service Factory registration.

The updated(String,Dictionary) method may throw a ConfigurationException object. This object describes the problem and what property caused the problem. These exceptions should be logged by a Configuration Admin service.

Multiple Managed Service Factory services can be registered with the same PID. Each of those services that have visibility to the corresponding configuration will be updated in service ranking order.

104.6.3 Deletion

If a configuring bundle deletes an instance of a Managed Service Factory, the deleted(String) method is called. The argument is the PID for this instance. The implementation of the Managed Service Factory must remove all information and stop any behavior associated with that PID. If a service was registered for this PID, it should be unregistered.

Deletion will asynchronously send out a Configuration Event CM_DELETED to all registered Configuration Listener services.

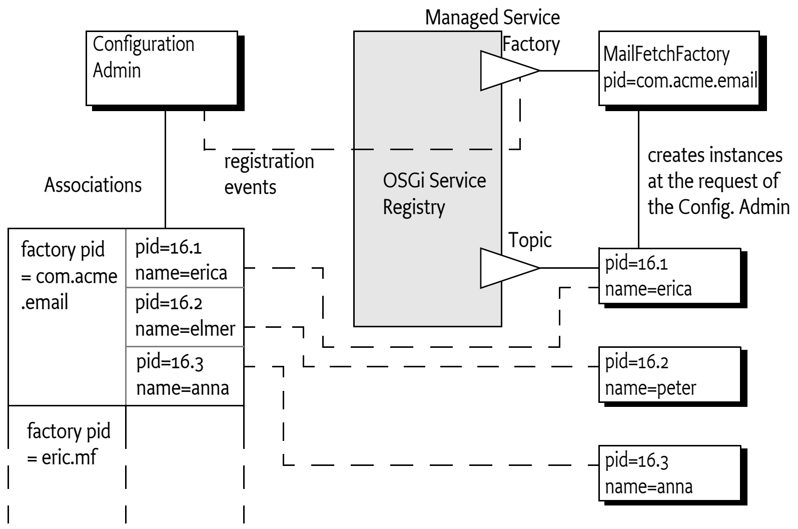
104.6.4 Managed Service Factory Example

Figure 104.7 highlights the differences between a Managed Service and a Managed Service Factory. It shows how a Managed Service Factory implementation receives configuration information that was created before it was registered.

- A bundle implements an EMail Fetcher service. It registers a ManagedServiceFactory object with PID=com.acme.email.
- The Configuration Admin service notices the registration and consults its database. It finds three Configuration objects for which the factory PID is equal to com.acme.email. It must call updated(String,Dictionary) for each of these Configuration objects on the newly registered ManagedServiceFactory object.
- For each configuration dictionary received, the factory should create a new instance of a EMailFetcher object, one for erica (PID=16.1), one for anna (PID=16.3), and one for elmer (PID=16.2).
- The EMailFetcher objects are registered under the Topic interface so their results can be viewed by an online display.

  If the EMailFetcher object is registered, it may safely use the PID of the Configuration object because the Configuration Admin service must guarantee its suitability for this purpose.

Figure 104.7 Managed Service Factory Example



104.6.5 Multiple Consoles Example

This example illustrates how multiple consoles, each of which has its own port and interface can run simultaneously. This approach is very similar to the example for the Managed Service, but highlights the difference by allowing multiple consoles to be created.

```
class ExampleFactory implements ManagedServiceFactory{
    Hashtable        consoles = new Hashtable();
    BundleContext    context;
    public void start( BundleContext context )
        throws Exception {
        this.context = context;
        Hashtable local = new Hashtable();
        local.put(Constants.SERVICE_PID,"com.acme.console");
        context.registerService(
            ManagedServiceFactory.class.getName(),
            this,
            local );
    }

    public void updated( String pid, Dictionary config ){
        Console console = (Console) consoles.get(pid);
        if (console == null) {
            console = new Console(context);
            consoles.put(pid, console);
        }

        int port = getInt(config, "port", 2011);
        String network = getString(
            config,
            "network",
            null /*all*/
        );
        console.setPort(port, network);
    }

    public void deleted(String pid) {
        Console console = (Console) consoles.get(pid);
        if (console != null) {
            consoles.remove(pid);
            console.close();
        }
    }
}
```

104.7 Configuration Admin Service

The ConfigurationAdmin interface provides methods to maintain configuration data in an OSGi environment. This configuration information is defined by a number of Configuration objects associated with specific configuration targets. Configuration objects can be created, listed, modified, and deleted through this interface. Either a remote management system or the bundles configuring their own configuration information may perform these operations.

The ConfigurationAdmin interface has methods for creating and accessing Configuration objects for a Managed Service, as well as methods for managing new Configuration objects for a Managed Service Factory.

104.7.1 Creating a Managed Service Configuration Object

A bundle can create a new Managed Service Configuration object with ConfigurationAdmin.getConfiguration. No create method is offered because doing so could introduce race conditions between different bundles trying to create a Configuration object for the same Managed Service. The getConfiguration method must atomically create and persistently store an object if it does not yet exist.

Two variants of this method are:

- getConfiguration(String) - This method is used by a bundle with a given location to configure its *own* ManagedService objects. The argument specifies the PID of the targeted service.
- getConfiguration(String,String) - This method is used by a management bundle to configure *another* bundle. Therefore, this management bundle needs the right permission. The first argument

is the PID and the second argument is the location identifier of the targeted ManagedService object.

All Configuration objects have a method, getFactoryPid(), which in this case must return null because the Configuration object is associated with a Managed Service.

Creating a new Configuration object must *not* initiate a callback to the Managed Service updated method until the properties are set in the Configuration with the update method.

104.7.2 Creating a Managed Service Factory Configuration Object

The ConfigurationAdmin class provides two sets of methods to create a new Configuration for a Managed Service Factory. The first set delegates the creation of the unique PID to the Configuration Admin service. The second set allows the caller to influence the generation of the PID.

The ConfigurationAdmin class provides the following two methods which generate a unique PID when creating a new Configuration for a Managed Service Factory. A new, unique PID is created for the Configuration object by the Configuration Admin service. The scheme used for this PID is defined by the Configuration Admin service and is unrelated to the factory PID, which is chosen by the registering bundle.

- createFactoryConfiguration(String) - This method is used by a bundle with a given location to configure its own ManagedServiceFactory objects. The argument specifies the PID of the targeted ManagedServiceFactory object. This *factory PID* can be obtained from the returned Configuration object with the getFactoryPid() method.
- createFactoryConfiguration(String,String) - This method is used by a management bundle to configure another bundle's ManagedServiceFactory object. The first argument is the PID and the second is the location identifier of the targeted ManagedServiceFactory object. The *factory PID* can be obtained from the returned Configuration object with getFactoryPid method.

The ConfigurationAdmin class provides the following two methods allowing the caller to influence the generation of the PID when creating a new Configuration for a Managed Service Factory. The PID for the Configuration object is generated from the provided factory PID and the provided name by starting with the factory PID, appending a tilde ('~' \u007e), and then appending the name. The getFactoryConfiguration methods must atomically create and persistently store a Configuration object if it does not yet exist.

- getFactoryConfiguration(String,String) - This method is used by a bundle with a given location to configure its own ManagedServiceFactory objects. The first argument specifies the PID of the targeted ManagedServiceFactory object. This *factory PID* can be obtained from the returned Configuration object with the getFactoryPid() method. The second argument specifies the *name* of the factory configuration. The generated PID can be obtained from the returned Configuration object with the getPid() method.
- getFactoryConfiguration(String,String,String) - This method is used by a management bundle to configure another bundle's ManagedServiceFactory object. The first argument is the PID, the second argument is the name, and the third is the location identifier of the targeted ManagedServiceFactory object. The *factory PID* can be obtained from the returned Configuration object with getFactoryPid method. The generated PID can be obtained from the returned Configuration object with the getPid() method.

Creating a new Configuration must *not* initiate a callback to the Managed Service Factory updated method until the properties are set in the Configuration object with the update method.

104.7.3 Accessing Existing Configurations

The existing set of Configuration objects can be listed with listConfigurations(String). The argument is a String object with a filter expression. This filter expression has the same syntax as the Framework Filter class. For example:

```
(&(size=42)(service.factoryPid=*osgi*))
```

The Configuration Admin service must only return Configurations that are visible to the calling bundle, see *Location Binding* on page 71.

A single Configuration object is identified with a PID, and can be obtained with listConfigurations(String) if it is visible. null is returned in both cases when there are no visible Configuration objects.

The PIDs that are filtered on can be targeted PIDs, see *Targeted PIDs* on page 69.

104.7.4 Updating a Configuration

The process of updating a Configuration object is the same for Managed Services and Managed Service Factories. First, listConfigurations(String), getConfiguration(String) or getFactoryConfiguration(String,String) should be used to get a Configuration object. The properties can be obtained with Configuration.getProperties. When no update has occurred since this object was created, getProperties returns null.

New properties can be set by calling Configuration.update. The Configuration Admin service must first store the configuration information and then call all configuration targets that have visibility with the updated method: either the ManagedService.updated(Dictionary) or ManagedServiceFactory.updated(String,Dictionary) method. If a target service is not registered, the fresh configuration information must be given to the target when the configuration target service registers and it has visibility. Each update of the Configuration properties must update a counter in the Configuration object after the data has been persisted but before the target(s) have been updated and any events are sent out. This counter is available from the getChangeCount() method.

The update methods in Configuration objects are not executed synchronously with the related target services updated method. The updated method must be called asynchronously. The Configuration Admin service, however, must have updated the persistent storage before the update method returns.

The update methods must also asynchronously send out a Configuration Event CM_UPDATED to all registered Configuration Listeners.

Invoking the update(Dictionary) method results in Configuration Admin service blindly updating the Configuration object and performing the above outlined actions. This even happens if the updated set of properties is the same as the already existing properties in the Configuration object.

To optimize configuration updates if the caller does not know whether properties of a Configuration object have changed, the updateIfDifferent(Dictionary) method can be used. The provided dictionary is compared with the existing properties. If there is no change, no action is taken. If there is any change detected, updateIfDifferent(Dictionary) acts exactly as update(Dictionary). Properties are compared as follows:

- Scalars are compared using equals
- Arrays are compared using Arrays.equals
- Collections are compared using equals

The boolean result of updateIfDifferent(Dictionary) is true if the Configuration object has been updated.

If the Configuration object has the READ_ONLY attribute set, calling one of the update methods results in a ReadOnlyConfigurationException and the configuration is not changed.

104.7.5 Using Multi-Locations

Sharing configuration between different bundles can be done using multi-locations, see *Location Binding* on page 71. A multi-location for a Configuration enables this Configuration to be delivered to any bundle that has visibility to that configuration. It is also possible that Bundles are interested in multiple PIDs for one target service, for this reason they can register multiple PIDs for one service.

For example, a number of bundles require access to the URL of a remote host, associated with the PID com.acme.host. A manager, aware that this PID is used by different bundles, would need to specify a location for the Configuration that allows delivery to any bundle. A multi-location, any location starting with a question mark achieves this. The part after the question mark has only use if the system runs with security, it allows the implementation of regions, see *Regions* on page 84. In this example a single question mark is used because any Bundle can receive this Configuration. The manager's code could look like:

```
Configuration c = admin.getConfiguration("com.acme.host", "?" );
Hashtable ht = new Hashtable();
ht.put( "host", hostURL);
c.update(ht);
```

A Bundle interested in the host configuration would register a Managed Service with the following properties:

```
service.pid = [ "com.acme.host", "com.acme.system"]
```

The Bundle would be called back for both the com.acme.host and com.acme.system PID and must therefore discriminate between these two cases. This Managed Service therefore would have a callback like:

```
volatile URL url;
public void updated( Dictionary d ) {
  if ( d.get("service.pid").equals("com.acme.host"))
      this.url = new URL( d.get("host"));
  if ( d.get("service.pid").equals("com.acme.system"))
        ....
}
```

104.7.6 Regions

In certain cases it is necessary to isolate bundles from each other. This will require that the configuration can be separated in *regions*. Each region can then be configured by a separate manager that is only allowed to manage bundles in its own region. Bundles can then only see configurations from their own region. Such a region based system can only be achieved with Java security as this is the only way to place bundles in a sandbox. This section describes how the Configuration's location binding can be used to implement regions if Java security is active.

Regions are groups of bundles that share location information among each other but are not willing to share this information with others. Using the multi-locations, see *Location Binding* on page 71, and security it is possible to limit access to a Configuration by using a location name. A Bundle can only receive a Configuration when it has ConfigurationPermission[location name, TARGET]. It is therefore possible to create region by choosing a region name for the location. A management agent then requires ConfigurationPermission[?region-name, CONFIGURE] and a Bundle in the region requires ConfigurationPermission[?region-name, TARGET].

To implement regions, the management agent is required to use multi-locations; without the question mark a Configuration is only visible to a Bundle that has the exact location of the Configuration. With a multi-location, the Configuration is delivered to any bundle that has the appropriate permission. Therefore, if regions are used, no manager should have ConfigurationPermission[*, CONFIGURE] because it would be able to configure anybody. This permission would enable the manager to set the location to any region or set the location to null. All managers must be restricted to a permission like ConfigurationPermission[?com.acme.region.*,CONFIGURE]. The resource name for a Configuration Permission uses substring matching as in the OSGi Filter, this facility can be used to simplify the administrative setup and implement more complex sharing schemes.

For example, a management agent works for the region com.acme. It has the following permission:

```
ConfigurationPermission[?com.acme.*,CONFIGURE]
```

The manager requires multi-location updates for com.acme.* (the last full stop is required in this wildcarding). For the CONFIGURE action the question mark must be specified in the resource name. The bundles in the region have the permission:

```
ConfigurationPermission["?com.acme.alpha",TARGET]
```

The question mark must be specified for the TARGET permission. A management agent that needs to configure Bundles in a region must then do this as follows:

```
Configuration c = admin.getConfiguration("com.acme.host", "?com.acme.alpha" );
Hashtable ht = new Hashtable();
ht.put( "host", hostURL);
c.update(ht);
```

Another, similar, example with two regions:

- system
- application

There is only one manager that manages all bundles. Its permissions look like:

```
ConfigurationPermission[?system,CONFIGURE]
ConfigurationPermission[?application,CONFIGURE]
```

A Bundle in the application region can have the following permissions:

```
ConfigurationPermission[?application,TARGET]
```

This managed bundle therefore has only visibility to configurations in the application region.

104.7.7 Deletion

A Configuration object that is no longer needed can be deleted with Configuration.delete, which removes the Configuration object from the database. The database must be updated before the target service's updated or deleted method is called. Only services that have received the configuration dictionary before must be called.

If the target service is a Managed Service Factory, the factory is informed of the deleted Configuration object by a call to ManagedServiceFactory.deleted(String) method. It should then remove the associated *instance*. The ManagedServiceFactory.deleted(String) call must be done asynchronously with respect to Configuration.delete().

When a Configuration object of a Managed Service is deleted, ManagedService.updated is called with null for the properties argument. This method may be used for clean-up, to revert to default values, or to unregister a service. This method is called asynchronously from the delete method.

The delete method must also asynchronously send out a Configuration Event CM_DELETED to all registered Configuration Listeners.

If the Configuration object has the READ_ONLY attribute set, calling the delete method results in a ReadOnlyConfigurationException and the configuration is not deleted.

104.7.8 Updating a Bundle's Own Configuration

The Configuration Admin service specification does not distinguish between updates via a Management Agent and a bundle updating its own configuration information (as defined by its location).

Even if a bundle updates its own configuration information, the Configuration Admin service must callback the associated target service's updated method.

As a rule, to update its own configuration, a bundle's user interface should *only* update the configuration information and never its internal structures directly. This rule has the advantage that the events, from the bundle implementation's perspective, appear similar for internal updates, remote management updates, and initialization.

104.7.9 Configuration Attributes

The Configuration object supports attributes, similar to setting attributes on files in a file system. Currently only the READ_ONLY attribute is supported.

Attributes can be set by calling the addAttributes(ConfigurationAttribute...) method and listing the attributes to be added. In the same way attributes can be removed by calling removeAttributes(ConfigurationAttribute...). Each successful change in attributes is persisted.

A Bundle can only change the attributes if it has Configuration Permission with the ATTRIBUTE action. Otherwise a Security Exception is thrown.

The currently set attributes can be queried using the getAttributes() method.

104.8 Configuration Events

Configuration Admin can update interested parties of changes in its repository. The model is based on the white board pattern where Configuration Listener services are registered with the service registry.

There are two types of Configuration Listener services:

- ConfigurationListener - The default Configuration Listener receives events asynchronously from the method that initiated the event and on another thread.
- SynchronousConfigurationListener - A Synchronous Configuration Listener is guaranteed to be called on the same thread as the method call that initiated the event.

The Configuration Listener service will receive ConfigurationEvent objects if important changes take place. The Configuration Admin service must call the configurationEvent(ConfigurationEvent) method with such an event. Configuration Events must be delivered in order for each listener as they are generated. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 8*.

The ConfigurationEvent object carries a factory PID (getFactoryPid()) and a PID (getPid()). If the factory PID is null, the event is related to a Managed Service Configuration object, else the event is related to a Managed Service Factory Configuration object.

The ConfigurationEvent object can deliver the following events from the getType() method:

- CM_DELETED - The Configuration object is deleted.
- CM_UPDATED - The Configuration object is updated.
- CM_LOCATION_CHANGED - The location of the Configuration object changed.

The Configuration Event also carries the ServiceReference object of the Configuration Admin service that generated the event.

104.8.1 Event Admin Service and Configuration Change Events

Configuration events must be delivered asynchronously via the Event Admin service, if present. The topic of a configuration event must be:

```
org/osgi/service/cm/ConfigurationEvent/<eventtype>
```

The <event type> can be any of the following:

CM_DELETED
CM_UPDATED
CM_LOCATION_CHANGED

The properties of a configuration event are:

- cm.factoryPid - (String) The factory PID of the associated Configuration object, if the target is a Managed Service Factory. Otherwise not set.
- cm.pid - (String) The PID of the associated Configuration object.
- service - (ServiceReference) The Service Reference of the Configuration Admin service.
- service.id - (Long) The Configuration Admin service's ID.
- service.objectClass - (String[]) The Configuration Admin service's object class (which must include org.osgi.service.cm.ConfigurationAdmin)
- service.pid - (String) The Configuration Admin service's persistent identity, if set.

104.9 Configuration Plugin

The Configuration Admin service allows third-party applications to participate in the configuration process. Bundles that register a service object under a ConfigurationPlugin interface can process the configuration dictionary just before it reaches the configuration target service.

Plug-ins allow sufficiently privileged bundles to intercept configuration dictionaries just *before* they must be passed to the intended Managed Service or Managed Service Factory but *after* the properties are stored. The changes the plug-in makes are dynamic and must not be stored. The plug-in must only be called when an update takes place while it is registered and there is a valid dictionary. The plugin is not called when a configuration is deleted.

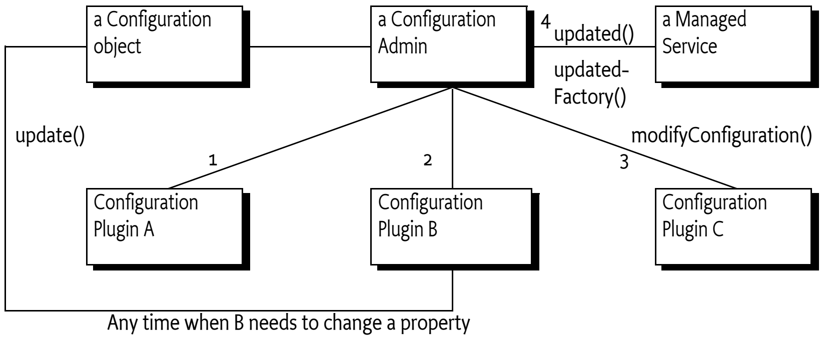
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.

All plug-ins in the service registry must be traversed and called before the properties are passed to the configuration target service. Each Configuration Plugin object gets a chance to inspect the existing data, look at the target object, which can be a ManagedService object or a ManagedServiceFactory object, and modify the properties of the configuration dictionary. The changes made by a plugin must be visible to plugins that are called later.

ConfigurationPlugin objects should not modify properties that belong to the configuration properties of the target service unless the implications are understood. This functionality is mainly intended to provide functions that leverage the Framework service registry. The changes made by the plugin should normally not be validated. However, the Configuration Admin must ignore changes to the automatic properties as described in *Automatic Properties* on page 73.

For example, a Configuration Plugin service may add a physical location property to a service. This property can be leveraged by applications that want to know where a service is physically located. This scenario could be carried out without any further support of the service itself, except for the general requirement that the service should propagate the public properties it receives from the Configuration Admin service to the service registry.

Figure 104.8 Order of Configuration Plugin Services



104.9.1 Limiting The Targets

A ConfigurationPlugin object may optionally specify a cm.target registration property. This value is the PID of the configuration target whose configuration updates the ConfigurationPlugin object wants to intercept.

The ConfigurationPlugin object must then only be called with updates for the configuration target service with the specified PID. For a factory target service, the factory PID is used and the plugin will see all instances of the factory. Omitting the cm.target registration property means that it is called for *all* configuration updates.

104.9.2 Example of Property Expansion

Consider a Managed Service that has a configuration property service.to with the value (objectclass=com.acme.Alarm). When the Configuration Admin service sets this property on the target service, a ConfigurationPlugin object may replace the (objectclass=com.acme.Alarm) filter with an array of existing alarm systems' PIDs as follows:

```
ID "service.to=[32434,232,12421,1212]"
```

A new Alarm Service with service.pid=343 is registered, requiring that the list of the target service be updated. The bundle which registered the Configuration Plugin service, therefore, wants to set the service.to registration property on the target service. It does *not* do this by calling ManagedService.updated directly for several reasons:

- In a securely configured system, it should not have the permission to make this call or even obtain the target service.
- It could get into race conditions with the Configuration Admin service if it had the permissions in the previous bullet. Both services would compete for access simultaneously.

Instead, it must get the Configuration object from the Configuration Admin service and call the update method on it.

The Configuration Admin service must schedule a new update cycle on another thread, and sometime in the future must call ConfigurationPlugin.modifyProperties. The ConfigurationPlugin object could then set the service.to property to [32434,232,12421,1212, 343]. After that, the Configuration Admin service must call updated on the target service with the new service.to list.

104.9.3 Configuration Data Modifications

Modifications to the configuration dictionary are still under the control of the Configuration Admin service, which must determine whether to accept the changes, hide critical variables, or deny the changes for other reasons.

The ConfigurationPlugin interface must also allow plugins to detect configuration updates to the service via the callback. This ability allows them to synchronize the configuration updates with transient information.

104.9.4 Forcing a Callback

If a bundle needs to force a Configuration Plugin service to be called again, it must fetch the appropriate Configuration object from the Configuration Admin service and call the update() method (the no parameter version) on this object. This call forces an update with the current configuration dictionary so that all applicable plug-ins get called again.

104.9.5 Calling Order

The order in which the ConfigurationPlugin objects are called must depend on the service.cmRanking configuration property of the ConfigurationPlugin object. Table 104.2 shows the usage of the service.cmRanking property for the order of calling the Configuration Plugin services. In the event of more than one plugin having the same value of service.cmRanking, then the order in which these are called is undefined.

Table 104.2 service.cmRanking Usage For Ordering

service.cmRanking value	Description
< 0	The Configuration Plugin service should not modify properties and must be called before any modifications are made. Any modification from the Configuration Plugin service is ignored.
>= 0 && <= 1000	The Configuration Plugin service modifies the configuration data. The calling order should be based on the value of the service.cmRanking property.
> 1000	The Configuration Plugin service should not modify data and is called after all modifications are made. Any modification from the Configuration Plugin service is ignored.

104.9.6 Manual Invocation

The Configuration Admin service ensures that Configuration Plugin services are automatically called for a Managed Service or a Managed Service Factory as outlined above. If a bundle needs to get the configuration properties processed by the Configuration Plugin services, the getProcessedProperties(ServiceReference) method provides this view.

The service reference passed into the method must either point to a Managed Service or Managed Service Factory registered on behalf of the bundle getting the processed properties. If that service should not be called by the Configuration Admin service, that service must be registered without a PID service property.

104.10 Meta Typing

This section discusses how the Metatype specification is used in the context of a Configuration Admin service.

When a Managed Service or Managed Service Factory is registered, the service object may also implement the MetaTypeProvider interface.

If the Managed Service or Managed Service Factory object implements the MetaTypeProvider interface, a management bundle may assume that the associated ObjectClassDefinition object can be used to configure the service.

The ObjectClassDefinition and AttributeDefinition objects contain sufficient information to automatically build simple user interfaces. They can also be used to augment dedicated interfaces with accurate validations.

When the Metatype specification is used, care should be taken to match the capabilities of the metatype package to the capabilities of the Configuration Admin service specification. Specifically:

- The metatype specification cannot describe nested arrays and lists or arrays/lists of mixed type.

This specification does not address how the metatype is made available to a management system due to the many open issues regarding remote management.

104.11 Coordinator Support

The *Coordinator Service Specification* on page 613 defines a mechanism for multiple parties to collaborate on a common task without *a priori* knowledge of who will collaborate in that task. The Configuration Admin service must participate in such scenarios to coordinate with provisioning or configuration tasks.

If configurations are created, updated or deleted and an implicit coordination exists, the Configuration Admin service must delay notifications until the coordination terminates. However the configuration changes must be persisted immediately. Updating a Managed Service or Managed Service Factory and informing asynchronous listeners is delayed until the coordination terminates, regardless of whether the coordination fails or terminates regularly. Registered synchronous listeners will be informed immediately when the change happens regardless of a coordination.

104.12 Capabilities

104.12.1 osgi.implementation Capability

The Configuration Admin implementation bundle must provide the osgi.implementation capability with the name osgi.cm. This capability can be used by provisioning tools and during resolution to ensure that a Configuration Admin implementation is present to manage configurations. The capability must also declare a uses constraint for the org.osgi.service.cm package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
       osgi.implementation="osgi.cm";
       uses:="org.osgi.service.cm";
       version:Version="1.6"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

Bundles relying on the Configuration Admin service should require the osgi.implementation capability from the Configuration Admin Service.

```
Require-Capability: osgi.implementation;
  filter:="(&(osgi.implementation=osgi.cm)(version>=1.6)(!(version>=2.0)))"
```

This requirement can be easily generated using the RequireConfigurationAdmin annotation.

104.12.2 osgi.service Capability

The bundle providing the Configuration Admin service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.cm package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.cm.ConfigurationAdmin";
  uses:="org.osgi.service.cm"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

104.13 Security

104.13.1 Configuration Permission

Every bundle has the implicit right to receive and configure configurations with a location that exactly matches the Bundle's location or that is null. For all other situations the Configuration Admin must verify that the configuring and to be updated bundles have a Configuration Permission that matches the Configuration's location.

The resource name of this permission maps to the location of the Configuration, the location can control the visibility of a Configuration for a bundle. The resource name is compared with the actual configuration location using the OSGi Filter sub-string matching. The question mark for multi-locations is part of the given resource name. The Configure Permission has the following actions:

- CONFIGURE - Can manage matching configurations
- TARGET - Can be updated with a matching configuration
- ATTRIBUTE - Can manage attributes for matching configuration

To be able to set the location to null requires a ConfigurationPermission[*, CONFIGURE].

It is possible to deny bundles the use of multi-locations by using Conditional Permission Admin's deny model.

104.13.2 Permissions Summary

Configuration Admin service security is implemented using Service Permission and Configuration Permission. The following table summarizes the permissions needed by the Configuration Admin bundle itself, as well as the typical permissions needed by the bundles with which it interacts.

Configuration Admin:

```
ServicePermission[ ..ConfigurationAdmin, REGISTER]
ServicePermission[ ..ManagedService, GET ]
ServicePermission[ ..ManagedServiceFactory, GET ]
ServicePermission[ ..ConfigurationPlugin, GET ]
ConfigurationPermission[ *, CONFIGURE ]
AdminPermission[ *, METADATA ]
```

Managed Service:

```
ServicePermission[ ..ConfigurationAdmin, GET]
ServicePermission[ ..ManagedService, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Managed Service Factory:

```
ServicePermission[ ..ConfigurationAdmin, GET]
ServicePermission[ ..ManagedServiceFactory, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Configuration Plugin:

```
ServicePermission[ ..ConfigurationPlugin,REGISTER ]
```

Configuration Listener:

```
ServicePermission[ ..ConfigurationListener,REGISTER ]
```

The Configuration Admin service must have ServicePermission[ConfigurationAdmin, REGISTER]. It will also be the only bundle that needs the ServicePermission[ManagedService | ManagedServiceFactory | ConfigurationPlugin, GET]. No other bundle should be allowed to have GET permission for these interfaces. The Configuration Admin bundle must also hold ConfigurationPermission[*,CONFIGURE].

Bundles that can be configured must have the ServicePermission[ManagedService | ManagedServiceFactory, REGISTER]. Bundles registering ConfigurationPlugin objects must have ServicePermission[ConfigurationPlugin, REGISTER]. The Configuration Admin service must trust all services registered with the ConfigurationPlugin interface. Only the Configuration Admin service should have ServicePermission[ConfigurationPlugin, GET].

If a Managed Service or Managed Service Factory is implemented by an object that is also registered under another interface, it is possible, although inappropriate, for a bundle other than the Configuration Admin service implementation to call the updated method. Security-aware bundles can avoid this problem by having their updated methods check that the caller has ConfigurationPermission[*,CONFIGURE].

Bundles that want to change their own configuration need ServicePermission[ConfigurationAdmin, GET]. A bundle with ConfigurationPermission[*,CONFIGURE] is allowed to access and modify any Configuration object.

Pre-configuration of bundles requires ConfigurationPermission[location,CONFIGURE] (location can use the sub-string matching rules of the Filter) because the methods that specify a location require this permission.

104.13.3 Configuration and Permission Administration

Configuration information has a direct influence on the permissions needed by a bundle. For example, when the Configuration Admin Bundle orders a bundle to use port 2011 for a console, that bundle also needs permission for listening to incoming connections on that port.

Both a simple and a complex solution exist for this situation.

The simple solution for this situation provides the bundle with a set of permissions that do not define specific values but allow a range of values. For example, a bundle could listen to ports above 1024 freely. All these ports could then be used for configuration.

The other solution is more complicated. In an environment where there is very strong security, the bundle would only be allowed access to a specific port. This situation requires an atomic update of both the configuration data and the permissions. If this update was not atomic, a potential security hole would exist during the period of time that the set of permissions did not match the configuration.

The following scenario can be used to update a configuration and the security permissions:

1. Stop the bundle.
2. Update the appropriate Configuration object via the Configuration Admin service.
3. Update the permissions in the Framework.
4. Start the bundle.

This scenario would achieve atomicity from the point of view of the bundle.

104.14 org.osgi.service.cm

Configuration Admin Package Version 1.6.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.cm; version="[1.6,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.cm; version="[1.6,1.7)"

104.14.1 Summary

- Configuration - The configuration information for a ManagedService or ManagedServiceFactory object.
- Configuration.ConfigurationAttribute - Configuration Attributes.
- ConfigurationAdmin - Service for administering configuration data.
- ConfigurationConstants - Defines standard constants for the Configuration Admin service.
- ConfigurationEvent - A Configuration Event.
- ConfigurationException - An Exception class to inform the Configuration Admin service of problems with configuration data.
- ConfigurationListener - Listener for Configuration Events.
- ConfigurationPermission - Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.
- ConfigurationPlugin - A service interface for processing configuration dictionary before the update.
- ManagedService - A service that can receive configuration data from a Configuration Admin service.
- ManagedServiceFactory - Manage multiple service instances.
- ReadOnlyConfigurationException - An Exception class to inform the client of a Configuration about the read only state of a configuration object.
- SynchronousConfigurationListener - Synchronous Listener for Configuration Events.

104.14.2 Permissions

104.14.2.1 Configuration

- setBundleLocation(String)
  - ConfigurationPermission[this.location,CONFIGURE] - if this.location is not null
  - ConfigurationPermission[location,CONFIGURE] - if location is not null
  - ConfigurationPermission["*",CONFIGURE] - if this.location is null or if location is null
- getBundleLocation()
  - ConfigurationPermission[this.location,CONFIGURE] - if this.location is not null
  - ConfigurationPermission["*",CONFIGURE] - if this.location is null
- addAttributes(ConfigurationAttribute...)
  - ConfigurationPermission[this.location,ATTRIBUTE] - if this.location is not null
  - ConfigurationPermission["*",ATTRIBUTE] - if this.location is null
- removeAttributes(ConfigurationAttribute...)

    - ConfigurationPermission[this.location,ATTRIBUTE] - if this.location is not null
    - ConfigurationPermission["*",ATTRIBUTE] - if this.location is null

104.14.2.2 ConfigurationAdmin

- createFactoryConfiguration(String,String)
    - ConfigurationPermission[location,CONFIGURE] - if location is not null
    - ConfigurationPermission["*",CONFIGURE] - if location is null
- getConfiguration(String,String)
    - ConfigurationPermission[*,CONFIGURE] - if location is null or if the returned configuration c already exists and c.location is null
    - ConfigurationPermission[location,CONFIGURE] - if location is not null
    - ConfigurationPermission[c.location,CONFIGURE] - if the returned configuration c already exists and c.location is not null
- getConfiguration(String)
    - ConfigurationPermission[c.location,CONFIGURE] - If the configuration c already exists and c.location is not null
- getFactoryConfiguration(String,String,String)
    - ConfigurationPermission[*,CONFIGURE] - if location is null or if the returned configuration c already exists and c.location is null
    - ConfigurationPermission[location,CONFIGURE] - if location is not null
    - ConfigurationPermission[c.location,CONFIGURE] - if the returned configuration c already exists and c.location is not null
- getFactoryConfiguration(String,String)
    - ConfigurationPermission[c.location,CONFIGURE] - If the configuration c already exists and c.location is not null
- listConfigurations(String)
    - ConfigurationPermission[c.location,CONFIGURE] - Only configurations c are returned for which the caller has this permission

104.14.2.3 ManagedService

- updated(Dictionary)
    - ConfigurationPermission[c.location,TARGET] - Required by the bundle that registered this service

104.14.2.4 ManagedServiceFactory

- updated(String,Dictionary)
    - ConfigurationPermission[c.location,TARGET] - Required by the bundle that registered this service

104.14.3 public interface Configuration

The configuration information for a ManagedService or ManagedServiceFactory object. The Configuration Admin service uses this interface to represent the configuration information for a ManagedService or for a service instance of a ManagedServiceFactory.

A Configuration object contains a configuration dictionary and allows the properties to be updated via this object. Bundles wishing to receive configuration dictionaries do not need to use this class - they register a ManagedService or ManagedServiceFactory. Only administrative bundles, and bundles wishing to update their own configurations need to use this class.

The properties handled in this configuration have case insensitive String objects as keys. However, case must be preserved from the last set key/value.

A configuration can be *bound* to a specific bundle or to a region of bundles using the *location*. In its simplest form the location is the location of the target bundle that registered a Managed Service or a Managed Service Factory. However, if the location starts with ? then the location indicates multiple delivery. In such a case the configuration must be delivered to all targets. If security is on, the Configuration Permission can be used to restrict the targets that receive updates. The Configuration Admin must only update a target when the configuration location matches the location of the target's bundle or the target bundle has a Configuration Permission with the action ConfigurationPermission.TARGET and a name that matches the configuration location. The name in the permission may contain wildcards ('*') to match the location using the same substring matching rules as Filter. Bundles can always create, manipulate, and be updated from configurations that have a location that matches their bundle location.

If a configuration's location is null, it is not yet bound to a location. It will become bound to the location of the first bundle that registers a ManagedService or ManagedServiceFactory object with the corresponding PID.

The same Configuration object is used for configuring both a Managed Service Factory and a Managed Service. When it is important to differentiate between these two the term "factory configuration" is used.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

104.14.3.1 public void addAttributes(Configuration.ConfigurationAttribute... attrs) throws IOException

attrs The attributes to add.

□ Add attributes to the configuration.

Throws IOException– If the new state cannot be persisted.

IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

Security ConfigurationPermission[this.location,ATTRIBUTE]] – if this.location is not null

ConfigurationPermission["*",ATTRIBUTE]] – if this.location is null

Since 1.6

104.14.3.2 public void delete() throws IOException

□ Delete this Configuration object.

Removes this configuration object from the persistent store. Notify asynchronously the corresponding Managed Service or Managed Service Factory. A ManagedService object is notified by a call to its updated method with a null properties argument. A ManagedServiceFactory object is notified by a call to its deleted method.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_DELETED event.

Throws ReadOnlyConfigurationException– If the configuration is read only.

IOException– If delete fails.

IllegalStateException– If this configuration has been deleted.

104.14.3.3 public boolean equals(Object other)

other Configuration object to compare against

□ Equality is defined to have equal PIDs Two Configuration objects are equal when their PIDs are equal.

Returns true if equal, false if not a Configuration object or one with a different PID.

104.14.3.4 public Set<Configuration.ConfigurationAttribute> getAttributes()

□ Get the attributes of this configuration.

Returns The set of attributes.

Throws IllegalStateException– If this configuration has been deleted.

Since 1.6

104.14.3.5 public String getBundleLocation()

□ Get the bundle location. Returns the bundle location or region to which this configuration is bound, or null if it is not yet bound to a bundle location or region. If the location starts with ? then the configuration is delivered to all targets and not restricted to a single bundle.

Returns location to which this configuration is bound, or null.

Throws IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

Security ConfigurationPermission[this.location,CONFIGURE]] – if this.location is not null

ConfigurationPermission["*",CONFIGURE]] – if this.location is null

104.14.3.6 public long getChangeCount()

□ Get the change count. Each Configuration must maintain a change counter that is incremented with a positive value every time the configuration is updated and its properties are stored. The counter must be incremented before the targets are updated and events are sent out.

Returns A monotonically increasing value reflecting changes in this Configuration.

Throws IllegalStateException– If this configuration has been deleted.

Since 1.5

104.14.3.7 public String getFactoryPid()

□ For a factory configuration return the PID of the corresponding Managed Service Factory, else return null.

Returns factory PID or null

Throws IllegalStateException– If this configuration has been deleted.

104.14.3.8 public String getPid()

□ Get the PID for this Configuration object.

Returns the PID for this Configuration object.

Throws IllegalStateException– if this configuration has been deleted

104.14.3.9 public Dictionary<String, Object> getProcessedProperties(ServiceReference<?> reference)

reference The reference to the Managed Service or Managed Service Factory to pass to the registered ConfigurationPlugins handling this configuration. Must not be null.

□ Return the processed properties of this Configuration object.

The Dictionary object returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type String.

Before the properties are returned they are processed by all the registered ConfigurationPlugins handling this configuration.

If called just after the configuration is created and before update has been called, this method returns null.

Returns A private copy of the processed properties for the caller or null. These properties must not contain the "service.bundleLocation" property. The value of this property may be obtained from the getBundleLocation() method.

Throws IllegalStateException– If this configuration has been deleted.

Since 1.6

104.14.3.10 public Dictionary<String, Object> getProperties()

□ Return the properties of this Configuration object. The Dictionary object returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type String.

If called just after the configuration is created and before update has been called, this method returns null.

Returns A private copy of the properties for the caller or null. These properties must not contain the "service.bundleLocation" property. The value of this property may be obtained from the getBundleLocation() method.

Throws IllegalStateException– If this configuration has been deleted.

104.14.3.11 public int hashCode()

□ Hash code is based on PID. The hash code for two Configuration objects must be the same when the Configuration PID's are the same.

Returns hash code for this Configuration object

104.14.3.12 public void removeAttributes(Configuration.ConfigurationAttribute... attrs) throws IOException

attrs The attributes to remove.

□ Remove attributes from this configuration.

Throws IOException– If the new state cannot be persisted.

IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

Security ConfigurationPermission[this.location,ATTRIBUTE]] – if this.location is not null

ConfigurationPermission["*",ATTRIBUTE]] – if this.location is null

Since 1.6

104.14.3.13 public void setBundleLocation(String location)

location a location, region, or null

□ Bind this Configuration object to the specified location. If the location parameter is null then the Configuration object will not be bound to a location/region. It will be set to the bundle's location before the first time a Managed Service/Managed Service Factory receives this Configuration object via the updated method and before any plugins are called. The bundle location or region will be set persistently.

If the location starts with ? then all targets registered with the given PID must be updated.

If the location is changed then existing targets must be informed. If they can no longer see this configuration, the configuration must be deleted or updated with null. If this configuration becomes visible then they must be updated with this configuration.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_LOCATION_CHANGED event.

Throws IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

Security ConfigurationPermission[this.location,CONFIGURE]] – if this.location is not null

ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission["*",CONFIGURE]] – if this.location is null or if location is null

104.14.3.14 public void update(Dictionary<String, ?> properties) throws IOException

properties the new set of properties for this configuration

□ Update the properties of this Configuration object.

Stores the properties in persistent storage after adding or overwriting the following properties:

- "service.pid" : is set to be the PID of this configuration.
- "service.factoryPid" : if this is a factory configuration it is set to the factory PID else it is not set.

These system properties are all of type String.

If the corresponding Managed Service/Managed Service Factory is registered, its updated method must be called asynchronously. Else, this callback is delayed until aforementioned registration occurs.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_UPDATED event.

Throws ReadOnlyConfigurationException– If the configuration is read only.

IOException– if update cannot be made persistent

IllegalArgumentException– if the Dictionary object contains invalid configuration types or contains case variants of the same key name.

IllegalStateException– If this configuration has been deleted.

104.14.3.15 public void update() throws IOException

□ Update the Configuration object with the current properties. Initiate the updated callback to the Managed Service or Managed Service Factory with the current properties asynchronously.

This is the only way for a bundle that uses a Configuration Plugin service to initiate a callback. For example, when that bundle detects a change that requires an update of the Managed Service or Managed Service Factory via its ConfigurationPlugin object.

Throws IOException– if update cannot access the properties in persistent storage

IllegalStateException– If this configuration has been deleted.

See Also ConfigurationPlugin

104.14.3.16 public boolean updateIfDifferent(Dictionary<String, ?> properties) throws IOException

properties The new set of properties for this configuration.

□ Update the properties of this Configuration object if the provided properties are different than the currently stored set. Properties are compared as follows.

- Scalars are compared using equals
- Arrays are compared using Arrays.equals
- Collections are compared using equals

If the new properties are not different than the current properties, no operation is performed. Otherwise, the behavior of this method is identical to the update(Dictionary) method.

Returns If the properties are different and the configuration is updated true is returned. If the properties are the same, false is returned.

Throws ReadOnlyConfigurationException – If the configuration is read only.

IOException – If update cannot be made persistent.

IllegalArgumentException – If the Dictionary object contains invalid configuration types or contains case variants of the same key name.

IllegalStateException – If this configuration has been deleted.

Since 1.6

104.14.4 enum Configuration.ConfigurationAttribute

Configuration Attributes.

Since 1.6

104.14.4.1 READ_ONLY

The configuration is read only.

104.14.4.2 public static Configuration.ConfigurationAttribute valueOf(String name)

104.14.4.3 public static Configuration.ConfigurationAttribute[] values()

104.14.5 public interface ConfigurationAdmin

Service for administering configuration data.

The main purpose of this interface is to store bundle configuration data persistently. This information is represented in Configuration objects. The actual configuration data is a Dictionary of properties inside a Configuration object.

There are two principally different ways to manage configurations. First there is the concept of a Managed Service, where configuration data is uniquely associated with an object registered with the service registry.

Next, there is the concept of a factory where the Configuration Admin service will maintain 0 or more Configuration objects for a Managed Service Factory that is registered with the Framework.

The first concept is intended for configuration data about "things/services" whose existence is defined externally, e.g. a specific printer. Factories are intended for "things/services" that can be created any number of times, e.g. a configuration for a DHCP server for different networks.

Bundles that require configuration should register a Managed Service or a Managed Service Factory in the service registry. A registration property named service.pid (persistent identifier or PID) must be used to identify this Managed Service or Managed Service Factory to the Configuration Admin service.

When the ConfigurationAdmin detects the registration of a Managed Service, it checks its persistent storage for a configuration object whose service.pid property matches the PID service property (service.pid) of the Managed Service. If found, it calls ManagedService.updated(Dictionary) method with the new properties. The implementation of a Configuration Admin service must run these callbacks asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose service.factoryPid property matches the PID service property of the Managed Service Factory. For each such Configuration objects, it calls the

ManagedServiceFactory.updated method asynchronously with the new properties. The calls to the updated method of a ManagedServiceFactory must be executed sequentially and not overlap in time.

In general, bundles having permission to use the Configuration Admin service can only access and modify their own configuration information. Accessing or modifying the configuration of other bundles requires ConfigurationPermission[location,CONFIGURE], where location is the configuration location.

Configuration objects can be *bound* to a specified bundle location or to a region (configuration location starts with ?). If a location is not set, it will be learned the first time a target is registered. If the location is learned this way, the Configuration Admin service must detect if the bundle corresponding to the location is uninstalled. If this occurs, the Configuration object must be unbound, that is its location field is set back to null.

If target's bundle location matches the configuration location it is always updated.

If the configuration location starts with ?, that is, the location is a region, then the configuration must be delivered to all targets registered with the given PID. If security is on, the target bundle must have Configuration Permission[location,TARGET], where location matches given the configuration location with wildcards as in the Filter substring match. The security must be verified using the org.osgi.framework.Bundle.hasPermission(Object) method on the target bundle.

If a target cannot be updated because the location does not match or it has no permission and security is active then the Configuration Admin service must not do the normal callback.

The method descriptions of this class refer to a concept of "the calling bundle". This is a loose way of referring to the bundle which obtained the Configuration Admin service from the service registry. Implementations of ConfigurationAdmin must use a org.osgi.framework.ServiceFactory to support this concept.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

104.14.5.1 public static final String SERVICE_BUNDLELOCATION = "service.bundleLocation"

Configuration property naming the location of the bundle that is associated with a Configuration object. This property can be searched for but must not appear in the configuration dictionary for security reason. The property's value is of type String.

Since 1.1

104.14.5.2 public static final String SERVICE_FACTORYPID = "service.factoryPid"

Configuration property naming the Factory PID in the configuration dictionary. The property's value is of type String.

Since 1.1

104.14.5.3 public Configuration createFactoryConfiguration(String factoryPid) throws IOException

factoryPid PID of factory (not null).

□ Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration object is bound to the location of the calling bundle. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

Returns A new Configuration object.

Throws IOException– if access to persistent storage fails.

104.14.5.4 public Configuration createFactoryConfiguration(String factoryPid, String location) throws IOException

factoryPid PID of factory (not null).

location A bundle location string, or null.

□ Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration is bound to the location specified. If this location is null it will be bound to the location of the first bundle that registers a Managed Service Factory with a corresponding PID. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

If the location starts with ? then the configuration must be delivered to all targets with the corresponding PID.

Returns a new Configuration object.

Throws IOException– if access to persistent storage fails.

SecurityException– when the require permissions are not available

Security ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission["*",CONFIGURE]] – if location is null

104.14.5.5 public Configuration getConfiguration(String pid, String location) throws IOException

pid Persistent identifier.

location The bundle location string, or null.

□ Get an existing Configuration object from the persistent store, or create a new Configuration object.

If a Configuration with this PID already exists in Configuration Admin service return it. The location parameter is ignored in this case though it is still used for a security check.

Else, return a new Configuration object. This new object is bound to the location and the properties are set to null. If the location parameter is null, it will be set when a Managed Service with the corresponding PID is registered for the first time. If the location starts with ? then the configuration is bound to all targets that are registered with the corresponding PID.

Returns An existing or new Configuration object.

Throws IOException– if access to persistent storage fails.

SecurityException– when the require permissions are not available

Security ConfigurationPermission[*,CONFIGURE]] – if location is null or if the returned configuration c already exists and c.location is null

ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission[c.location,CONFIGURE]] – if the returned configuration c already exists and c.location is not null

104.14.5.6 public Configuration getConfiguration(String pid) throws IOException

pid persistent identifier.

□ Get an existing or new Configuration object from the persistent store. If the Configuration object for this PID does not exist, create a new Configuration object for that PID, where properties are null. Bind its location to the calling bundle's location.

Otherwise, if the location of the existing Configuration object is null, set it to the calling bundle's location.

Returns an existing or new Configuration matching the PID.

Throws IOException– if access to persistent storage fails.

SecurityException– when the required permission is not available

Security ConfigurationPermission[c.location,CONFIGURE]] – If the configuration c already exists and c.location is not null

104.14.5.7 public Configuration getFactoryConfiguration(String factoryPid, String name, String location) throws IOException

factoryPid PID of factory (not null).

name A name for Configuration (not null).

location The bundle location string, or null.

□ Get an existing or new Configuration object from the persistent store. The PID for this Configuration object is generated from the provided factory PID and the name by starting with the factory PID appending a tilde ('~' \u007E), and then appending the name.

If a Configuration with this PID already exists in Configuration Admin service return it. The location parameter is ignored in this case though it is still used for a security check.

Else, return a new Configuration object. This new object is bound to the location and the properties are set to null. If the location parameter is null, it will be set when a Managed Service with the corresponding PID is registered for the first time. If the location starts with ? then the configuration is bound to all targets that are registered with the corresponding PID.

Returns An existing or new Configuration object.

Throws IOException– if access to persistent storage fails.

SecurityException– when the require permissions are not available

Security ConfigurationPermission[*,CONFIGURE]] – if location is null or if the returned configuration c already exists and c.location is null

ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission[c.location,CONFIGURE]] – if the returned configuration c already exists and c.location is not null

Since 1.6

104.14.5.8 public Configuration getFactoryConfiguration(String factoryPid, String name) throws IOException

factoryPid PID of factory (not null).

name A name for Configuration (not null).

□ Get an existing or new Configuration object from the persistent store. The PID for this Configuration object is generated from the provided factory PID and the name by starting with the factory PID appending a tilde ('~' \u007E), and then appending the name.

If a Configuration object for this PID does not exist, create a new Configuration object for that PID, where properties are null. Bind its location to the calling bundle's location.

Otherwise, if the location of the existing Configuration object is null, set it to the calling bundle's location.

Returns an existing or new Configuration matching the PID.

Throws IOException– if access to persistent storage fails.

SecurityException– when the required permission is not available

Security ConfigurationPermission[c.location,CONFIGURE]] – If the configuration c already exists and c.location is not null

Since 1.6

104.14.5.9 public Configuration[] listConfigurations(String filter) throws IOException, InvalidSyntaxException

filter A filter string, or null to retrieve all Configuration objects.

□ List the current Configuration objects which match the filter.

Only Configuration objects with non- null properties are considered current. That is, Configuration.getProperties() is guaranteed not to return null for each of the returned Configuration objects.

When there is no security on then all configurations can be returned. If security is on, the caller must have ConfigurationPermission[location,CONFIGURE].

The syntax of the filter string is as defined in the Filter class. The filter can test any configuration properties including the following:

- service.pid - the persistent identity
- service.factoryPid - the factory PID, if applicable
- service.bundleLocation - the bundle location

The filter can also be null, meaning that all Configuration objects should be returned.

Returns All matching Configuration objects, or null if there aren't any.

Throws IOException– if access to persistent storage fails

InvalidSyntaxException– if the filter string is invalid

Security ConfigurationPermission[c.location,CONFIGURE]] – Only configurations c are returned for which the caller has this permission

104.14.6 public final class ConfigurationConstants

Defines standard constants for the Configuration Admin service.

104.14.6.1 public static final String CONFIGURATION_ADMIN_IMPLEMENTATION = "osgi.cm"

The name of the implementation capability for the Configuration Admin specification

Since 1.6

104.14.6.2 public static final String CONFIGURATION_ADMIN_SPECIFICATION_VERSION = "1.6"

The version of the implementation capability for the Configuration Admin specification

Since 1.6

104.14.7 public class ConfigurationEvent

A Configuration Event.

ConfigurationEvent objects are delivered to all registered ConfigurationListener service objects. ConfigurationEvents must be delivered in chronological order with respect to each listener.

A type code is used to identify the type of event. The following event types are defined:

- CM_UPDATED
- CM_DELETED
- CM_LOCATION_CHANGED

Additional event types may be defined in the future.

Security Considerations. ConfigurationEvent objects do not provide Configuration objects, so no sensitive configuration information is available from the event. If the listener wants to locate the Configuration object for the specified pid, it must use ConfigurationAdmin.

See Also ConfigurationListener

Since 1.2

Concurrency Immutable

104.14.7.1 public static final int CM_DELETED = 2

A Configuration has been deleted.

This ConfigurationEvent type that indicates that a Configuration object has been deleted. An event is fired when a call to Configuration.delete() successfully deletes a configuration.

104.14.7.2 public static final int CM_LOCATION_CHANGED = 3

The location of a Configuration has been changed.

This ConfigurationEvent type that indicates that the location of a Configuration object has been changed. An event is fired when a call to Configuration.setBundleLocation(String) successfully changes the location.

Since 1.4

104.14.7.3 public static final int CM_UPDATED = 1

A Configuration has been updated.

This ConfigurationEvent type that indicates that a Configuration object has been updated with new properties. An event is fired when a call to Configuration.update(Dictionary) successfully changes a configuration.

104.14.7.4 public ConfigurationEvent(ServiceReference<ConfigurationAdmin> reference, int type, String factoryPid, String pid)

reference The ServiceReference object of the Configuration Admin service that created this event.

type The event type. See getType().

factoryPid The factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise null if the target of the configuration is a ManagedService.

pid The pid of the associated configuration.

□ Constructs a ConfigurationEvent object from the given ServiceReference object, event type, and pids.

104.14.7.5 public String getFactoryPid()

□ Returns the factory pid of the associated configuration.

Returns Returns the factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise null if the target of the configuration is a ManagedService.

104.14.7.6 public String getPid()

□ Returns the pid of the associated configuration.

Returns Returns the pid of the associated configuration.

104.14.7.7 public ServiceReference<ConfigurationAdmin> getReference()

□ Return the ServiceReference object of the Configuration Admin service that created this event.

Returns The ServiceReference object for the Configuration Admin service that created this event.

104.14.7.8 public int getType()

□ Return the type of this event.

The type values are:

- CM_UPDATED
- CM_DELETED
- CM_LOCATION_CHANGED

Returns The type of this event.

104.14.8 public class ConfigurationException extends Exception

An Exception class to inform the Configuration Admin service of problems with configuration data.

104.14.8.1 public ConfigurationException(String property, String reason)

property name of the property that caused the problem, null if no specific property was the cause

reason reason for failure

□ Create a ConfigurationException object.

104.14.8.2 public ConfigurationException(String property, String reason, Throwable cause)

property name of the property that caused the problem, null if no specific property was the cause

reason reason for failure

cause The cause of this exception.

□ Create a ConfigurationException object.

Since 1.2

104.14.8.3 public Throwable getCause()

□ Returns the cause of this exception or null if no cause was set.

Returns The cause of this exception or null if no cause was set.

Since 1.2

104.14.8.4 public String getProperty()

□ Return the property name that caused the failure or null.

Returns name of property or null if no specific property caused the problem

104.14.8.5 public String getReason()

□ Return the reason for this exception.

Returns reason of the failure

104.14.8.6 public Throwable initCause(Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws IllegalArgumentException– If the specified cause is this exception.

IllegalStateException– If the cause of this exception has already been set.

Since 1.2

104.14.9 public interface ConfigurationListener

Listener for Configuration Events. When a ConfigurationEvent is fired, it is asynchronously delivered to all ConfigurationListeners.

ConfigurationListener objects are registered with the Framework service registry and are notified with a ConfigurationEvent object when an event is fired.

ConfigurationListener objects can inspect the received ConfigurationEvent object to determine its type, the pid of the Configuration object with which it is associated, and the Configuration Admin service that fired the event.

Security Considerations. Bundles wishing to monitor configuration events will require ServicePermission[ConfigurationListener,REGISTER] to register a ConfigurationListener service.

Since 1.2

Concurrency Thread-safe

104.14.9.1 public void configurationEvent(ConfigurationEvent event)

event The ConfigurationEvent.

□ Receives notification of a Configuration that has changed.

104.14.10 public final class ConfigurationPermission extends BasicPermission

Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.

Since 1.2

Concurrency Thread-safe

104.14.10.1 public static final String ATTRIBUTE = "attribute"

Provides permission to set or remove an attribute on the configuration. The action string "attribute".

Since 1.6

104.14.10.2 public static final String CONFIGURE = "configure"

Provides permission to create new configurations for other bundles as well as manipulate them. The action string "configure".

104.14.10.3 public static final String TARGET = "target"

The permission to be updated, that is, act as a Managed Service or Managed Service Factory. The action string "target".

Since 1.4

104.14.10.4 public ConfigurationPermission(String name, String actions)

name Name of the permission. Wildcards ('*') are allowed in the name. During implies(Permission), the name is matched to the requested permission using the substring matching rules used by Filters.

actions Comma separated list of CONFIGURE, TARGET, ATTRIBUTE (case insensitive).

□ Create a new ConfigurationPermission.

104.14.10.5 public boolean equals(Object obj)

obj The object being compared for equality with this object.

□ Determines the equality of two ConfigurationPermission objects.

Two ConfigurationPermission objects are equal.

Returns true if obj is equivalent to this ConfigurationPermission; false otherwise.

104.14.10.6 public String getActions()

□ Returns the canonical string representation of the ConfigurationPermission actions.

Always returns present ConfigurationPermission actions in the following order: "configure", "target", "attribute".

Returns Canonical string representation of the ConfigurationPermission actions.

104.14.10.7 public int hashCode()

□ Returns the hash code value for this object.

Returns Hash code value for this object.

104.14.10.8 public boolean implies(Permission p)

p The target permission to check.

□ Determines if a ConfigurationPermission object "implies" the specified permission.

Returns true if the specified permission is implied by this object; false otherwise.

104.14.10.9 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing ConfigurationPermissions.

Returns A new PermissionCollection object.

104.14.11 public interface ConfigurationPlugin

A service interface for processing configuration dictionary before the update.

A bundle registers a ConfigurationPlugin object in order to process configuration updates before they reach the Managed Service or Managed Service Factory. The Configuration Admin service will detect registrations of Configuration Plugin services and must call these services every time before it calls the ManagedService or ManagedServiceFactory updated method. The Configuration Plugin service thus has the opportunity to view and modify the properties before they are passed to the Managed Service or Managed Service Factory.

Configuration Plugin (plugin) services have full read/write access to all configuration information that passes through them.

The Integer service.cmRanking registration property may be specified. Not specifying this registration property, or setting it to something other than an Integer, is the same as setting it to the Integer zero. The service.cmRanking property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones. In the event of more than one plugin having the same value of service.cmRanking, then the Configuration Admin service arbitrarily chooses the order in which they are called.

By convention, plugins with service.cmRanking < 0 or service.cmRanking > 1000 should not make modifications to the properties. Any modifications made by such plugins must be ignored.

The Configuration Admin service has the right to hide properties from plugins, or to ignore some or all the changes that they make. This might be done for security reasons. Any such behavior is entirely implementation defined.

A plugin may optionally specify a cm.target registration property whose value is the PID of the Managed Service or Managed Service Factory whose configuration updates the plugin is intended to intercept. The plugin will then only be called with configuration updates that are targeted at the Managed Service or Managed Service Factory with the specified PID. Omitting the cm.target registration property means that the plugin is called for all configuration updates.

Concurrency Thread-safe

104.14.11.1 public static final String CM_RANKING = "service.cmRanking"

A service property to specify the order in which plugins are invoked. This property contains an Integer ranking of the plugin. Not specifying this registration property, or setting it to something other than an Integer, is the same as setting it to the Integer zero. This property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones.

Since 1.2

104.14.11.2 public static final String CM_TARGET = "cm.target"

A service property to limit the Managed Service or Managed Service Factory configuration dictionaries a Configuration Plugin service receives. This property contains a String[] of PIDs. A Configuration Admin service must call a Configuration Plugin service only when this property is not set, or the target service's PID is listed in this property.

104.14.11.3 public void modifyConfiguration(ServiceReference<?> reference, Dictionary<String, Object> properties)

reference reference to the Managed Service or Managed Service Factory

properties The configuration properties. This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the Configuration.getBundleLocation method.

□ View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory. The Configuration Plugin services are called in increasing order of their service.cmRanking property. If this property is undefined or is a non- Integer type, 0 is used.

This method should not modify the properties unless the service.cmRanking of this plugin is in the range 0 <= service.cmRanking <= 1000. Any modification from this plugin is ignored.

If this method throws any Exception, the Configuration Admin service must catch it and should log it. Any modifications made by the plugin before the exception is thrown are applied.

A Configuration Plugin will only be called for properties from configurations that have a location for which the Configuration Plugin has permission when security is active. When security is not active, no filtering is done.

104.14.12 public interface ManagedService

A service that can receive configuration data from a Configuration Admin service.

A Managed Service is a service that needs configuration data. Such an object should be registered with the Framework registry with the service.pid property set to some unique identifier called a PID.

If the Configuration Admin service has a Configuration object corresponding to this PID, it will callback the updated() method of the ManagedService object, passing the properties of that Configuration object.

If it has no such Configuration object, then it calls back with a null properties argument. Registering a Managed Service will always result in a callback to the updated() method provided the Configuration Admin service is, or becomes active. This callback must always be done asynchronously.

Else, every time that either of the updated() methods is called on that Configuration object, the ManagedService.updated() method with the new properties is called. If the delete() method is called on that Configuration object, ManagedService.updated() is called with a null for the properties parameter. All these callbacks must be done asynchronously.

The following example shows the code of a serial port that will create a port depending on configuration information.

```
class SerialPort implements ManagedService {

  ServiceRegistration registration;
  Hashtable configuration;
  CommPortIdentifier id;

  synchronized void open(CommPortIdentifier id,
  BundleContext context) {
    this.id = id;
    registration = context.registerService(
      ManagedService.class.getName(),
      this,
      getDefaults()
    );
  }

  Hashtable getDefaults() {
    Hashtable defaults = new Hashtable();
    defaults.put( "port", id.getName() );
    defaults.put( "product", "unknown" );
    defaults.put( "baud", "9600" );
    defaults.put( Constants.SERVICE_PID,
      "com.acme.serialport." + id.getName() );
    return defaults;
  }

  public synchronized void updated(
    Dictionary configuration  ) {
    if ( configuration == null )
      registration.setProperties( getDefaults() );
    else {
      setSpeed( configuration.get("baud") );
      registration.setProperties( configuration );
    }
  }
  ...
}
```

As a convention, it is recommended that when a Managed Service is updated, it should copy all the properties it does not recognize into the service registration properties. This will allow the Configuration Admin service to set properties on services which can then be used by other applications.

Normally, a single Managed Service for a given PID is given the configuration dictionary, this is the configuration that is bound to the location of the registering bundle. However, when security is on, a Managed Service can have Configuration Permission to also be updated for other locations.

If a Managed Service is registered without the service.pid property, it will be ignored.

Concurrency Thread-safe

104.14.12.1 public void updated(Dictionary<String, ?> properties) throws ConfigurationException

properties A copy of the Configuration properties, or `null`. This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

□ Update the configuration for a Managed Service.

When the implementation of `updated(Dictionary)` detects any kind of error in the configuration properties, it should create a new `ConfigurationException` which describes the problem. This can allow a management system to provide useful information to a human administrator.

If this method throws any other `Exception`, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously with the method that initiated the callback. This implies that implementors of Managed Service can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the location allows multiple managed services to be called back for a single configuration then the callbacks must occur in service ranking order. Changes in the location must be reflected by deleting the configuration if the configuration is no longer visible and updating when it becomes visible.

If no configuration exists for the corresponding PID, or the bundle has no access to the configuration, then the bundle must be called back with a `null` to signal that CM is active but there is no data.

Throws `ConfigurationException` – when the update fails

Security `ConfigurationPermission[c.location,TARGET]]` – Required by the bundle that registered this service

104.14.13 public interface ManagedServiceFactory

Manage multiple service instances. Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.

Each of these *service instances* is represented, in the persistent storage of the Configuration Admin service, by a factory `Configuration` object that has a PID. When such a `Configuration` is updated, the Configuration Admin service calls the `ManagedServiceFactory` updated method with the new properties. When `updated` is called with a new PID, the Managed Service Factory should create a new factory instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information.

In general it is expected that the implementation of this interface will maintain a data structure that maps PIDs to the factory instances that it has created. The semantics of a factory instance are defined by the Managed Service Factory. However, if the factory instance is registered as a service object with the service registry, its PID should match the PID of the corresponding `Configuration` object (but it should **not** be registered as a Managed Service!).

An example that demonstrates the use of a factory. It will create serial ports under command of the Configuration Admin service.

```
class SerialPortFactory
  implements ManagedServiceFactory {
  ServiceRegistration registration;
  Hashtable ports;
  void start(BundleContext context) {
    Hashtable properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      "com.acme.serialportfactory" );
    registration = context.registerService(
      ManagedServiceFactory.class.getName(),
      this,
      properties
    );
  }
  public void updated( String pid,
    Dictionary properties  ) {
    String portName = (String) properties.get("port");
    SerialPortService port =
      (SerialPort) ports.get( pid );
    if ( port == null ) {
      port = new SerialPortService();
      ports.put( pid, port );
      port.open();
    }
    if ( port.getPortName().equals(portName) )
      return;
    port.setPortName( portName );
  }
  public void deleted( String pid ) {
    SerialPortService port =
      (SerialPort) ports.get( pid );
    port.close();
    ports.remove( pid );
  }
  ...
}
```

If a ManagedServiceFactory is registered without the service.pid property, it will be ignored.

Concurrency Thread-safe

104.14.13.1 public void deleted(String pid)

pid the PID of the service to be removed

□ Remove a factory instance. Remove the factory instance associated with the PID. If the instance was registered with the service registry, it should be unregistered. The Configuration Admin must call deleted for each instance it received in updated(String, Dictionary).

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously.

104.14.13.2 public String getName()

□ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

104.14.13.3 public void updated(String pid, Dictionary<String, ?> properties) throws ConfigurationException

pid The PID for this configuration.

properties A copy of the configuration properties. This argument must not contain the service.bundleLocation" property. The value of this property may be obtained from the Configuration.getBundleLocation method.

□ Create a new instance, or update the configuration of an existing instance. If the PID of the Configuration object is new for the Managed Service Factory, then create a new factory instance, using the configuration properties provided. Else, update the service instance with the provided properties.

If the factory instance is registered with the Framework, then the configuration properties should be copied to its registry properties. This is not mandatory and security sensitive properties should obviously not be copied.

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

When the implementation of updated detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem.

The Configuration Admin service must call this method asynchronously. This implies that implementors of the ManagedServiceFactory class can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the security allows multiple managed service factories to be called back for a single configuration then the callbacks must occur in service ranking order.

It is valid to create multiple factory instances that are bound to different locations. Managed Service Factory services must only be updated with configurations that are bound to their location or that start with the ? prefix and for which they have permission. Changes in the location must be reflected by deleting the corresponding configuration if the configuration is no longer visible or updating when it becomes visible.

Throws ConfigurationException– when the configuration properties are invalid.

Security ConfigurationPermission[c.location,TARGET]] – Required by the bundle that registered this service

104.14.14 public class ReadOnlyConfigurationException extends RuntimeException

An Exception class to inform the client of a Configuration about the read only state of a configuration object.

Since 1.6

104.14.14.1 public ReadOnlyConfigurationException(String reason)

reason reason for failure

□ Create a ReadOnlyConfigurationException object.

104.14.15 public interface SynchronousConfigurationListener extends ConfigurationListener

Synchronous Listener for Configuration Events. When a ConfigurationEvent is fired, it is synchronously delivered to all SynchronousConfigurationListeners.

SynchronousConfigurationListener objects are registered with the Framework service registry and are synchronously notified with a ConfigurationEvent object when an event is fired.

SynchronousConfigurationListener objects can inspect the received ConfigurationEvent object to determine its type, the PID of the Configuration object with which it is associated, and the Configuration Admin service that fired the event.

Security Considerations. Bundles wishing to synchronously monitor configuration events will require ServicePermission[SynchronousConfigurationListener,REGISTER] to register a SynchronousConfigurationListener service.

Since 1.5

Concurrency Thread-safe

104.15 org.osgi.service.cm.annotations

Configuration Admin Annotations Package Version 1.6.

This package contains annotations that can be used to require the Configuration Admin implementations

Bundles should not normally need to import this package as the annotations are only used at build-time.

104.15.1 Summary

- `RequireConfigurationAdmin` - This annotation can be used to require the Configuration Admin implementation.

104.15.2 @RequireConfigurationAdmin

This annotation can be used to require the Configuration Admin implementation. It can be used directly, or as a meta-annotation.

Since 1.6

Retention `CLASS`

Target `TYPE`, `PACKAGE`

105 Metatype Service Specification

Version 1.4

105.1 Introduction

The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called *metadata*.

The purpose of this specification is to allow services to specify the type information of data that they can use as arguments. The data is based on *attributes*, which are key/value pairs like properties.

A designer in a type-safe language like Java is often confronted with the choice of using the language constructs to exchange data or using a technique based on attributes/properties that are based on key/value pairs. Attributes provide an escape from the rigid type-safety requirements of modern programming languages.

Type-safety works very well for software development environments in which multiple programmers work together on large applications or systems, but often lacks the flexibility needed to receive structured data from the outside world.

The attribute paradigm has several characteristics that make this approach suitable when data needs to be communicated between different entities which "speak" different languages. Attributes are uncomplicated, resilient to change, and allow the receiver to dynamically adapt to different types of data.

As an example, the OSGi framework Specifications define several attribute types which are used in a Framework implementation, but which are also used and referenced by other OSGi specifications such as the *Configuration Admin Service Specification* on page 65. A Configuration Admin service implementation deploys attributes (key/value pairs) as configuration properties.

The Meta Type Service provides a unified access point to the Meta Type information that is associated with bundles. This Meta Type information can be defined by an XML resource in a bundle (`OSGI-INF/metatype` directories must be scanned for any XML resources), it can come from the Meta Type Provider service, or it can be obtained from Managed Service or Managed Service Factory services.

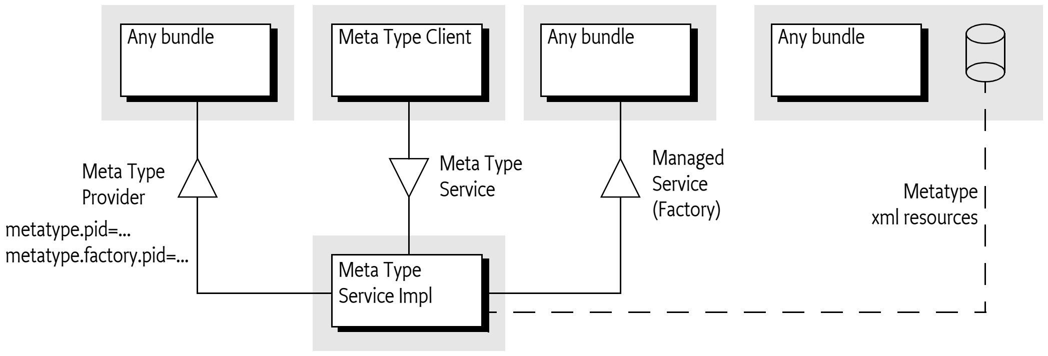
105.1.1 Essentials

- *Conceptual model* - The specification must have a conceptual model for how classes and attributes are organized.
- *Standards* - The specification should be aligned with appropriate standards, and explained in situations where the specification is not aligned with, or cannot be mapped to, standards.
- *Remote Management* - Remote management should be taken into account.
- *Size* - Minimal overhead in size for a bundle using this specification is required.
- *Localization* - It must be possible to use this specification with different languages at the same time. This ability allows servlets to serve information in the language selected in the browser.
- *Type information* - The definition of an attribute should contain the name (if it is required), the cardinality, a label, a description, labels for enumerated values, and the Java class that should be used for the values.
- *Validation* - It should be possible to validate the values of the attributes.

105.1.2 Entities

- *Meta Type Service* - A service that provides a unified access point for meta type information.
- *Attribute* - A key/value pair.
- *PID* - A unique persistent ID, defined in configuration management.
- *Attribute Definition* - Defines a description, name, help text, and type information of an attribute.
- *Object Class Definition* - Defines the type of a datum. It contains a description and name of the type plus a set of `AttributeDefinition` objects.
- *Meta Type Provider* - Provides access to the object classes that are available for this object. Access uses the PID and a locale to find the best `ObjectClassDefinition` object.
- *Meta Type Information* - Provides meta type information for a bundle.

Figure 105.1 Class Diagram Meta Type Service, org.osgi.service.metatype



105.1.3 Operation

The Meta Type service defines a rich dynamic typing system for properties. The purpose of the type system is to allow reasonable User Interfaces to be constructed dynamically.

The type information is normally carried by the bundles themselves. Either by implementing the `MetaTypeProvider` interface on the Managed Service or Managed Service Factory, by carrying one or more XML resources that define a number of Meta Types in the `OSGI-INF/metatype` directories, or registering a Meta Type Provider as a service. Additionally, a Meta Type service could have other sources that are not defined in this specification.

The Meta Type Service provides unified access to Meta Types that are carried by the resident bundles. The Meta Type Service collects this information from the bundles and provides uniform access to it. A client can requests the Meta Type Information associated with a particular bundle. The `MetaTypeInformation` object provides a list of `ObjectClassDefinition` objects for a bundle. These objects define all the information for a specific *object class*. An object class is a some descriptive information and a set of named attributes (which are key/value pairs).

Access to Object Class Definitions is qualified by a locale and a Persistent IDentity (PID). This specification does not specify what the PID means. One application is OSGi Configuration Management where a PID is used by the Managed Service and Managed Service Factory services. In general, a PID should be regarded as the name of a variable where an Object Class Definition defines its type.

105.2 Attributes Model

The Framework uses the LDAP filter syntax for searching the Framework registry. The usage of the attributes in this specification and the Framework specification closely resemble the LDAP attribute model. Therefore, the names used in this specification have been aligned with LDAP. Consequently, the interfaces which are defined by this Specification are:

- AttributeDefinition
- ObjectClassDefinition
- MetaTypeProvider

These names correspond to the LDAP attribute model. For further information on ASN.1-defined attributes and X.500 object classes and attributes, see [2] *Understanding and Deploying LDAP Directory services.*

The LDAP attribute model assumes a global name-space for attributes, and object classes consist of a number of attributes. So, if an object class inherits the same attribute from different parents, only one copy of the attribute must become part of the object class definition. This name-space implies that a given attribute, for example cn, should *always* be the common name and the type must always be a String. An attribute cn cannot be an Integer in another object class definition. In this respect, the OSGi approach towards attribute definitions is comparable with the LDAP attribute model.

105.3 Object Class Definition

The ObjectClassDefinition interface is used to group the attributes which are defined in AttributeDefinition objects.

An ObjectClassDefinition object contains the information about the overall set of attributes and has the following elements:

- A name which can be returned in different locales.
- A global name-space in the registry, which is the same condition as LDAP/X.500 object classes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organizations, and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned. This id can be a Java class name (reverse domain name) or can be generated with a GUID algorithm. All LDAP-defined object classes already have an associated OID. It is strongly advised to define the object classes from existing LDAP schemes which provide many preexisting OIDs. Many such schemes exist ranging from postal addresses to DHCP parameters.
- A human-readable description of the class.
- A list of attribute definitions which can be filtered as required, or optional. Note that in X.500 the mandatory or required status of an attribute is part of the object class definition and not of the attribute definition.
- An icon, in different sizes.

105.4 Attribute Definition

The AttributeDefinition interface provides the means to describe the data type of attributes.

The AttributeDefinition interface defines the following elements:

- Defined names (final ints) for the data types as restricted in the Framework for the attributes, called the syntax in OSI terms, which can be obtained with the getType() method.
- AttributeDefinition objects should use an ID that is similar to the OID as described in the ID field for ObjectClassDefinition.
- A localized name intended to be used in user interfaces.

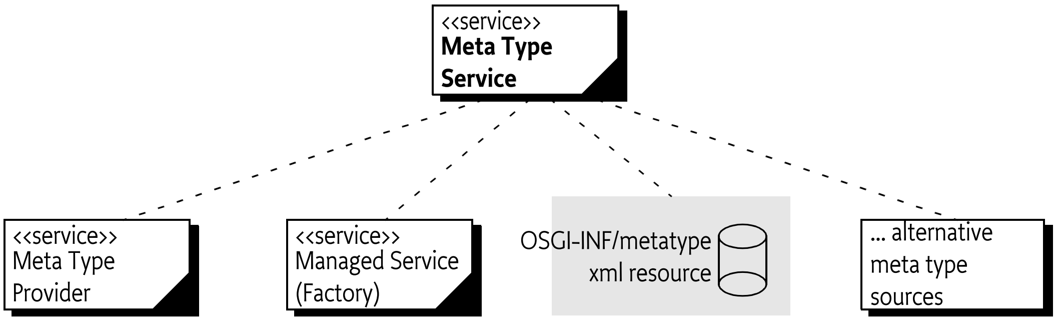
- A localized description that defines the semantics of the attribute and possible constraints, which should be usable for tooltips.
- An indication if this attribute should be stored as a unique value, a List, or an array of values, as well as the maximum cardinality of the type.
- The data type, as limited by the Framework service registry attribute types.
- A validation function to verify if a possible value is correct.
- A list of values and a list of localized labels. Intended for popup menus in GUIs, allowing the user to choose from a set.
- A default value (String[]). The return depends on the following cases:
  - *not specified* - Return null if this attribute is not specified.
  - *cardinality = 0* - Return an array with one element.
  - *otherwise* - Return an array with less or equal than the absolute value of cardinality, possibly empty if the value is an empty string.

105.5 Meta Type Service

The Meta Type Service provides unified access to Meta Type information that is associated with a Bundle. It can get this information through the following means:

- *Meta Type Resource* - A bundle can provide one or more XML resources that are contained in its JAR file. These resources contain an XML definition of meta types as well as to what PIDs these Meta Types apply. These XML resources must reside in the OSGI-INF/metatype directories of the bundle (including any fragments).
- *Managed Service [Factory] objects* - As defined in the configuration management specification, ManagedService and ManagedServiceFactory service objects can optionally implement the MetaTypeProvider interface. The Meta Type Service will only search for ManagedService and ManagedServiceFactory service objects that implement MetaTypeProvider if no meta type resources are found in the bundle.
- *Meta Type Provider service* - Bundles can register Meta Type Provider services to dynamically provide meta types for PIDs and factory PIDs.

Figure 105.2 Sources for Meta Types


<<service>>
Meta Type Service
<<service>>
Meta Type Provider
<<service>>
Managed Service (Factory)
OSGI-INF/metatype xml resource
... alternative meta type sources

This model is depicted in Figure 105.2.

The Meta Type Service can therefore be used to retrieve meta type information for bundles which contain Meta Type resources or which provide MetaTypeProvider objects and/or services. If multiple sources define the same Object Class Definition, the Meta Type service must select which source to use. Meta Type Provider services must take precedence over Managed Service [Factory] objects implementing MetaTypeProvider or Meta Type Resources.

The MetaTypeService interface has a single method:

- getMetaTypeInformation(Bundle) - Given a bundle, it must return the Meta Type Information for that bundle, even if there is no meta type information available at the moment of the call.

The returned MetaTypeInformation object maintains a map of PID to ObjectClassDefinition objects. The map is keyed by locale and PID. The list of maintained PIDs is available from the MetaTypeInformation object with the following methods:

- getPids() - PIDs for which Meta Types are available.
- getFactoryPids() - PIDs associated with Managed Service Factory services.

These methods and their interaction with the Meta Type resource are described in *Designate Element* on page 124.

The MetaTypeInformation interface extends the MetaTypeProvider interface. The MetaType-Provider interface is used to access meta type information. It supports locale dependent information so that the text used in AttributeDefinition and ObjectClassDefinition objects can be adapted to different locales.

Which locales are supported by the MetaTypeProvider object are defined by the implementer or the meta type resources. The list of available locales can be obtained from the MetaTypeProvider object.

The MetaTypeProvider interface provides the following methods:

- getObjectClassDefinition(String,String) - Get access to an ObjectClassDefinition object for the given PID. The second parameter defines the locale.
- getLocales() - List the locales that are available.

Locale objects are represented in String objects because not all profiles support Locale. The String holds the standard Locale presentation of:

```
locale = language ( '_' country ( '_' variation))
language ::= < defined by ISO 3166 >
country  ::= < defined by ISO 639 >
```

For example, en, nl_BE, en_CA_posix are valid locales. The use of null for locale indicates that java.util.Locale.getDefault() must be used.

The Meta Type Service implementation class is the main class. It registers the org.osgi.service.metatype.MetaTypeService service and has a method to get a MetaTypeInformation object for a bundle.

Following is some sample code demonstrating how to print out all the Object Class Definitions and Attribute Definitions contained in a bundle:

```
void printMetaTypes( MetaTypeService mts,Bundle b ) {
    MetaTypeInformation mti =
        mts.getMetaTypeInformation(b);
    String [] pids = mti.getPids();
    String [] locales = mti.getLocales();

    for ( int locale = 0; locale<locales.length; locale++) {
        System.out.println("Locale " + locales[locale] );
        for (int i=0; i< pids.length; i++) {
            ObjectClassDefinition ocd =
                mti.getObjectClassDefinition(pids[i], null);
            AttributeDefinition[] ads =
                ocd.getAttributeDefinitions(
                    ObjectClassDefinition.ALL);
            for (int j=0; j< ads.length; j++) {
                System.out.println("OCD="+ocd.getName()
                    + "AD="+ads[j].getName());
            }
        }
    }
}
```

105.6 Meta Type Provider Service

A Meta Type Provider service allows third party contributions to the internal Object Class Definition repository. A Meta Type Provider can contribute multiple PIDs, both factory and singleton PIDs. A Meta Type Provider service must register with both or one of the following service properties:

- METATYPE_PID - (String+) Provides a list of PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as normal singleton PIDs used by Managed Services.
- METATYPE_FACTORY_PID - (String+) Provides a list of factory PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as factory PIDs used by Managed Service Factories.

The Object Class Definitions must originate from the bundle that registered the Meta Type Provider service. Third party extenders should therefore use the bundle of their extendee. A Meta Type Service must report these Object Class Definitions on the Meta Type Information of the registering bundle, merged with any other information from that bundle.

The Meta Type Service must track these Meta Type Provider services and make their Meta Types available as if they were provided on the Managed Service (Factory) services. The Meta Types must become unavailable when the Meta Type Provider service is unregistered.

105.7 Using the Meta Type Resources

A bundle that wants to provide meta type resources must place these resources in the OSGI-INF/metatype directory. The name of the resource must be a valid bundle entry path. All resources in that directory must be meta type documents. Resources in that directory that are not valid meta type documents must be ignored and an error should be logged with the Log Service, if present. Fragments can contain additional meta type resources in the same directory and they must be taken into account when the meta type resources are searched. A meta type resource must be encoded in UTF-8.

The MetaType Service must support localization of the

- name
- icon
- description
- label attributes

The localization mechanism must be identical using the same mechanism as described in the Core module layer, see *Localization*, using the same property resource. However, it is possible to override the property resource in the meta type definition resources with the localization attribute of the MetaData element.

The Meta Type Service must examine the bundle and its fragments to locate all localization resources for the localization base name. From that list, the Meta Type Service derives the list of locales which are available for the meta type information. This list can then be returned by MetaTypeInformation.getLocales method. This list can change at any time because the bundle could be refreshed. Clients should be prepared that this list changes after they received it.

105.7.1 XML Schema of a Meta Type Resource

This section describes the schema of the meta type resource. This schema is not intended to be used during runtime for validating meta type resources. The schema is intended to be used by tools and external management systems.

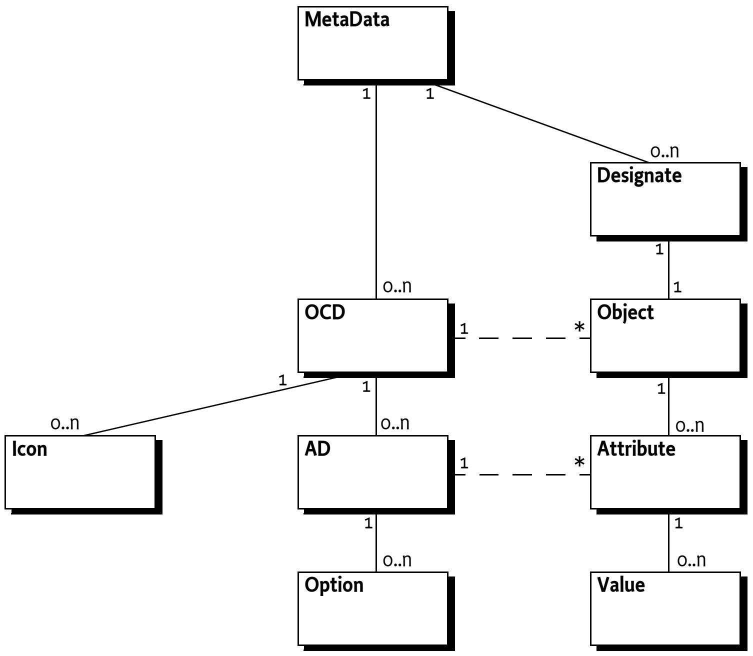
The XML namespace for meta type documents must be:

```
http://www.osgi.org/xmlns/metatype/v1.4.0
```

The namespace abbreviation should be metatype. That is, the following header should be:

```
<metatype:MetaData
    xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.4.0">
```

Figure 105.3 *XML Schema Instance Structure (Type name = Element name)*



The element structure of the XML file is:

```
MetaData    ::= OCD* Designate*

OCD         ::= AD*  Icon*
AD          ::= Option*

Designate   ::= Object
Object      ::= Attribute*

Attribute   ::= Value*
```

The different elements are described in Table 105.1.

Table 105.1 XML Schema for Meta Type resources

Attribute	Deflt	Type	Method	Description
MetaData				Top Element
localization		string		Points to the Properties file that can localize this XML. See *Localization* in *OSGi Core Release 8*.
OCD				Object Class Definition
name	<>	string	getName()	A human readable name that can be localized.
description			getDescription()	A human readable description of the Object Class Definition that can be localized.
id	<>		getID()	A unique id, cannot be localized.
Designate				An association between one PID and an Object Class Definition. This element *designates* a PID to be of a certain *type*.
pid	<>	string		The PID that is associated with an OCD. This can be a reference to a factory or singleton configuration object. The PID can be a Targeted PID, if factoryPid is not set or empty. Either pid or factoryPid must be specified. See *Designate Element* on page 124.
factoryPid		string		If the factoryPid attribute is set, this Designate element defines a factory configuration for the given factory. If it is not set or empty, it designates a singleton configuration. The PID can be a Targeted PID. Either pid or factoryPid must be specified. See *Designate Element* on page 124.
bundle		string		The value is used to set the location of any configuration created using this Meta Type resource. This may contain a bundle location or a multi-location. In a Meta Type resource, using the wildcard value ('*' \u002A) indicates the bundle location of the bundle containing the resource must be used as the location. See *Location Binding* on page 71 This is an optional attribute but can be mandatory in certain usage schemes, for example the Autoconf Resource Processor.
optional	false	boolean		If true, then this Designate element is optional, errors during processing must be ignored.
merge	false	boolean		If the PID refers to an existing configuration, then merge the properties with the existing properties if this attribute is true. Otherwise, replace the properties.

Attribute	Deflt	Type	Method	Description
AD				Attribute Definition
name		string	getName()	A localizable name for the Attribute Definition. description
description		string	getDescription()	A localizable description for the Attribute Definition.
id			getID()	The unique ID of the Attribute Definition.
type		string	getType()	The type of an attribute is an enumeration of the different scalar types. The string is mapped to one of the constants on the AttributeDefinition interface. Valid values, which are defined in the Scalar type, are:
				String ↔ STRING
				Long ↔ LONG
				Double ↔ DOUBLE
				Float ↔ FLOAT
				Integer ↔ INTEGER
				Byte ↔ BYTE
				Char ↔ CHARACTER
				Boolean ↔ BOOLEAN
				Short ↔ SHORT
				Password ↔ PASSWORD
cardinality	0		getCardinality()	The number of elements an instance can take. Positive numbers describe an array ([]) and negative numbers describe a List object.
min		string	validate(String)	A validation value. This value is not directly available from the AttributeDefinition interface. However, the validate(String) method must verify this. The semantics of this field depend on the type of this Attribute Definition.
max		string	validate(String)	A validation value. Similar to the min field. When min or max are numbers, attribute values with a numeric data type are valid if min <= value <= max. Attribute values with a string (or equivalent) data type are valid if min <= value.length() <= max.

Attribute	Deflt	Type	Method	Description
default		string	getDefaultValue()	The default value. A default is an array of String objects. The XML attribute must contain a comma delimited list. The default value is trimmed and escaped in the same way as described in the validate(String) method. The empty string is significant and must be seen as an empty List or array if specified as the default for an attribute with a cardinality that is not equal to zero. Default values must be valid or otherwise ignored.
required	true	boolean		Required attribute. The required attribute indicates whether or not the attribute key must appear within the configuration dictionary to be valid.
Option				One option label/value for the options in an AD. Options are exclusive. The validate(String) method must verify that an attribute value matches one of the option values when present.
label	<>	string	getOptionLabels()	The label
value	<>	string	getOptionValues()	The value
Icon				An icon definition.
resource	<>	string	getIcon(int)	The resource is a URL. The base URL is assumed to be the root of the bundle containing the XML file. That is, this URL can reference another resource in the bundle using a relative URL.
size	<>	string	getIcon(int)	The number of pixels of the icon, maps to the size parameter of the getIcon(int) method.
Object				A definition of an instance.
ocdref	<>	string		A reference to the id attribute of an OCD element. That is, this attribute defines the OCD type of this object.
Attribute				A value for an attribute of an object.
adref	<>	string		A reference to the id of the AD in the OCD as referenced by the parent Object.
content		string		The content of the attributes. If this is an array, the content must be separated by commas (',' \u002C). Commas must be escaped as described at the default attribute of the AD element.
Value				Holds a single value. This element can be repeated multiple times under an Attribute

105.7.2 Designate Element

For the MetaType Service, the Designate definition is used to declare the available PIDs and factory PIDs; the Attribute elements are never used by the MetaType service.

The getPids() method returns an array of PIDs that were specified in the pid attribute of the Object elements. The getFactoryPids() method returns an array of the factoryPid attributes. For factories, the related pid attribute is ignored because all instances of a factory must share the same meta type.

The following example shows a metatype reference to a singleton configuration and a factory configuration.

```
    <Designate pid="com.acme.designate.1">
        <Object ocdref="com.acme.designate"/>
    </Designate>
    <Designate factoryPid="com.acme.designate.factory"
        bundle="*">
        <Object ocdref="com.acme.designate"/>
    </Designate>
```

Other schemes can embed the Object element in the Designate element to define actual instances for the Configuration Admin service. In that case the pid attribute must be used together with the factoryPid attribute. However, in that case an aliasing model is required because the Configuration Admin service does not allow the creator to choose the Configuration object's PID.

105.7.3 Example Metadata File

This example defines a meta type file for a Person record, based on ISO attribute types. The ids that are used are derived from ISO attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData
    xmlns="http://www.osgi.org/xmlns/metatype/v1.4.0"
    localization="person">
  <OCD name="%person" id="2.5.6.6"
        description="%person record">
    <AD name="%sex" id="2.5.4.12" type="Integer">
       <Option label="%male" value="1"/>
       <Option label="%female" value="0"/>
    </AD>
    <AD name="%sn" id="2.5.4.4" type="String"/>
    <AD name="%cn" id="2.5.4.3" type="String"/>
    <AD name="%seeAlso" id="2.5.4.34" type="String"
        cardinality="8"
        default="http://www.google.com,http://www.yahoo.com"/>
    <AD name="%telNumber" id="2.5.4.20" type="String"/>
  </OCD>

  <Designate pid="com.acme.addressbook">
    <Object ocdref="2.5.6.6"/>
  </Designate>
</MetaData>
```

Translations for this file, as indicated by the localization attribute must be stored in the root directory (e.g. person_du_NL.properties). The default localization base name for the properties is OSGI-INF/l10n/bundle, but can be overridden by the manifest Bundle-Localization header and the localization attribute of the Meta Data element. The property files have the base name of person. The Dutch, French and English translations could look like:

person_du_NL.properties:

```
person=Persoon
person\ record=Persoons beschrijving
```

```
cn=Naam
sn=Voornaam
seeAlso=Zie ook
telNumber=Tel. Nummer
sex=Geslacht
male=Mannelijk
female=Vrouwelijk
```

person_fr.properties:

```
person=Personne
person\ record=Description de la personne
cn=Nom
sn=Surnom
seeAlso=Reference
telNumber=Tel.
sex=Sexe
male=Homme
female=Femme
```

person_en_US.properties:

```
person=Person
person\ record=Person Record
cn=Name
sn=Sur Name
seeAlso=See Also
telNumber=Tel.
sex=Sex
male=Male
female=Female
```

105.7.4 Object Element

The OCD element can be used to describe the possible contents of a Dictionary object. In this case, the attribute name is the key. The Object element can be used to assign a value to a Dictionary object.

For example:

```
<Designate pid="com.acme.b">
  <Object ocdref="b">
    <Attribute adref="foo" content="Zaphod Beeblebrox"/>
    <Attribute adref="bar">
      <Value>1</Value>
      <Value>2</Value>
      <Value>3</Value>
      <Value>4</Value>
      <Value>5</Value>
    </Attribute>
  </Object>
</Designate>
```

105.8 Meta Type Resource XML Schema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.4.0"
    targetNamespace="http://www.osgi.org/xmlns/metatype/v1.4.0"
    version="1.4.0">

    <element name="MetaData" type="metatype:Tmetadata" />

    <complexType name="Tmetadata">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="OCD" type="metatype:Tocd" />
            <element name="Designate" type="metatype:Tdesignate" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="localization" type="string" use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tocd">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="AD" type="metatype:Tad" />
            <element name="Icon" type="metatype:Ticon" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="name" type="string" use="required" />
        <attribute name="description" type="string" use="optional" />
        <attribute name="id" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tad">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Option" type="metatype:Toption" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax"  />
        </choice>
        <attribute name="name" type="string" use="optional" />
        <attribute name="description" type="string" use="optional" />
        <attribute name="id" type="string" use="required" />
        <attribute name="type" type="metatype:Tscalar" use="required" />
        <attribute name="cardinality" type="int" use="optional"
            default="0" />
        <attribute name="min" type="string" use="optional" />
        <attribute name="max" type="string" use="optional" />
        <attribute name="default" type="string" use="optional" />
        <attribute name="required" type="boolean" use="optional"
            default="true" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tobject">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Attribute" type="metatype:Tattribute" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="ocdref" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tattribute">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Value" type="string" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="adref" type="string" use="required" />
        <attribute name="content" type="string" use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tdesignate">
        <sequence>
            <element name="Object" type="metatype:Tobject" minOccurs="1"
                maxOccurs="1" />
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="pid" type="string" use="optional" />
        <attribute name="factoryPid" type="string" use="optional" />
        <attribute name="bundle" type="string" use="optional" />
        <attribute name="optional" type="boolean" default="false"
            use="optional" />
        <attribute name="merge" type="boolean" default="false"
            use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>

    <simpleType name="Tscalar">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="Float" />
            <enumeration value="Integer" />
            <enumeration value="Byte" />
            <enumeration value="Character" />
            <enumeration value="Boolean" />
            <enumeration value="Short" />
            <enumeration value="Password" />
        </restriction>
    </simpleType>

    <complexType name="Toption">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="label" type="string" use="required" />
        <attribute name="value" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Ticon">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="resource" type="string" use="required" />
        <attribute name="size" type="positiveInteger" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <attribute name="must-understand" type="boolean">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents
                to require that the document consumer understand the
                extension.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

105.9 Meta Type Annotations

A developer can use Meta Type Annotations on a Component Property Type, see *Component Property Types* on page 292, or an interface to define an Object Class Definition in a type safe manner. The Meta Type Annotations are CLASS retention annotations intended to be used during build time to generate Meta Type Resources from the Java class files providing a convenient way to create the Meta Type Resource XML documents.

Tools processing these annotations must always generate valid Meta Type Resource XML documents. If the Meta Type Annotations are used in a way that is not supported or in error, then the tool must report the error to enable the developer to take corrective action.

105.9.1 ObjectClassDefinition Annotation

The ObjectClassDefinition annotation can be applied to a Component Property Type or an interface. From that type, tooling can generate an OCD element. When applied to an interface, all the methods inherited from supertypes are include as Attribute Definitions. The tool processing the annotations must be able to examine all the types in the hierarchy of the annotated type to generate the Meta Type Resource. It is an error if the tool cannot examine a type in the hierarchy.

It is an error to apply the ObjectClassDefinition annotation to concrete and abstract class types. It is also an error to apply it to an interface if any of the methods of the interface take arguments.

The ObjectClassDefinition annotation can be applied without defining any element values as default values for the ObjectClassDefinition annotation elements can be generated from the annotated type. For example:

```
@ObjectClassDefinition
@interface Config {
  boolean enabled();
  String[] names();
  String topic();
}
```

In the following larger example, the ObjectClassDefinition annotation defines the description and name of the OCD which are to be localized using the specified resource as well as an icon resource. Also, AttributeDefinition annotations are applied to the methods to supply some non-default values for the generated AD elements.

```
@ObjectClassDefinition(localization = "OSGI-INF/l10n/member",
    description = "%member.description",
    name = "%member.name",
    icon = @Icon(resource = "icon/member-32.png", size = 32))
@interface Member {
  @AttributeDefinition(type = AttributeType.PASSWORD,
    description = "%member.password.description",
    name = "%member.password.name")
  public String _password();

  @AttributeDefinition(options = {
    @Option(label = "%strategic", value = "strategic"),
    @Option(label = "%principal", value = "principal"),
    @Option(label = "%contributing", value = "contributing")
    },
    defaultValue = "contributing",
    description = "%member.membertype.description",
    name = "%member.membertype.name")
  public String type();
}
```

105.9.2 AttributeDefinition Annotation

The AttributeDefinition annotation is an optional annotation which can applied to the methods in a type annotated by ObjectClassDefinition. Each method of the type annotated by ObjectClassDefinition is mapped to an AD child element of the OCD element in the generated Meta Type Resource XML document. The AttributeDefinition annotation only needs to be applied to a method if values other than the defaults are desired.

The id of the Attribute Definition is generated from the method name as follows:

- A single dollar sign ('$' \u0024) is removed unless it is followed by:
  - A low line ('_' \u005F) and a dollar sign in which case the three consecutive characters ("$_$") are converted to a single hyphen-minus ('-' \u002D).
  - Another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.
- A single low line ('_' \u005F) is converted into a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.
- If the type declaring the method also declares a PREFIX_ field whose value is a compile-time constant String, then the id is prefixed with the value of the PREFIX_ field.

However, if the type annotated by ObjectClassDefinition is a *single-element annotation*, see 9.7.3 in [3] *The Java Language Specification, Java SE 8 Edition*, then the id for the value method is derived from the name of the annotation type rather than the name of the method. In this case, the simple name of the annotation type, that is, the name of the class without any package name or outer class name, if the annotation type is an inner class, must be converted to the value method's id as follows:

- When a lower case character is followed by an upper case character, a full stop ('.' \u002E) is inserted between them.
- Each upper case character is converted to lower case.
- All other characters are unchanged.
- If the annotation type declares a PREFIX_ field whose value is a compile-time constant String, then the id is prefixed with the value of the PREFIX_ field.

The generated id becomes the value of the id attribute of the AD element in the generated Meta Type Resource XML document.

105.9.3 Designate Annotation

The Designate annotation can be applied to a Declarative Services component class to make the connection between the pid of the component and an Object Class Definition. This annotation must be used on a type that is also annotated with the Declarative Services Component annotation. The component must only have a single PID which is used for the generated Designate element.

In the following example, the Designate annotation is applied to a Declarative Services component and references the Object Class Definition type.

```
@ObjectClassDefinition(id="my.config.ocd")
@interface Config {
  boolean enabled() default true;
  String[] names() default {"a", "b"};
  String topic() default "default/topic";
}
```

```
@Component(configurationPid="my.component.pid")
@Designate(ocd = Config.class)
public class MyComponent {
    static final String DEFAULT_TOPIC_PREFIX = "topic.prefix";
    protected void activate(Config configuration) {
        String t = configuration.topic();
    }
}
```

Tools processing these annotations will generate a Designate element in the generated Meta Type Resource XML document using the PID of the component and the id of the Object Class Definition. For example:

```
<Designate pid="my.component.pid">
    <Object ocdref="my.config.ocd"/>
</Designate>
```

105.10 Limitations

The OSGi MetaType specification is intended to be used for simple applications. It does not, therefore, support recursive data types, mixed types in arrays/lists, or nested arrays/lists.

105.11 Related Standards

One of the primary goals of this specification is to make metatype information available at run-time with minimal overhead. Many related standards are applicable to metatypes; except for Java beans, however, all other metatype standards are based on document formats (e.g. XML). In the OSGi framework, document format standards are deemed unsuitable due to the overhead required in the execution environment (they require a parser during run-time).

Another consideration is the applicability of these standards. Most of these standards were developed for management systems on platforms where resources are not necessarily a concern. In this case, a metatype standard is normally used to describe the data structures needed to control some other computer via a network. This other computer, however, does not require the metatype information as it is *implementing* this information.

In some traditional cases, a management system uses the metatype information to control objects in an OSGi framework. Therefore, the concepts and the syntax of the metatype information must be mappable to these popular standards. Clearly, then, these standards must be able to describe objects in an OSGi framework. This ability is usually not a problem, because the metatype languages used by current management systems are very powerful.

105.12 Capabilities

Implementations of the Metatype Service specification must provide the following capabilities.

- A capability in the osgi.implementation namespace declaring a specification implementation with the name METATYPE_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

  ```
  Provide-Capability: osgi.implementation;
  ```

```
    osgi.implementation="osgi.metatype";
    version:Version="1.4";
    uses:="org.osgi.service.metatype"
```

The RequireMetaTypeImplementation annotation can be used to require this capability.

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

- A capability in the osgi.extender namespace declaring an extender with the name METATYPE_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

```
Provide-Capability: osgi.extender;
    osgi.extender="osgi.metatype";
    version:Version="1.4";
    uses:="org.osgi.service.metatype"
```

  The RequireMetaTypeExtender annotation can be used to require this capability.

  This capability must follow the rules defined for the *osgi.extender Namespace* on page 707.
- A capability in the osgi.service namespace representing the MetaTypeService service. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

```
Provide-Capability: osgi.service;
    objectClass:List<String>="org.osgi.service.metatype.MetaTypeService";
    uses:="org.osgi.service.metatype"
```

  This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

105.13 Security Considerations

Special security issues are not applicable for this specification.

105.14 org.osgi.service.metatype

Metatype Package Version 1.4.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.metatype; version="[1.4,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.metatype; version="[1.4,1.5)"

105.14.1 Summary

- AttributeDefinition - An interface to describe an attribute.
- MetaTypeInformation - A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.
- MetaTypeProvider - Provides access to metatypes.

- MetaTypeService - The MetaType Service can be used to obtain meta type information for a bundle.
- ObjectClassDefinition - Description for the data type information of an objectclass.

105.14.2 public interface AttributeDefinition

An interface to describe an attribute.

An AttributeDefinition object defines a description of the data type of a property/attribute.

Concurrency Thread-safe

105.14.2.1 public static final int BIGDECIMAL = 10

The BIGDECIMAL type. Attributes of this type should be stored as BigDecimal, List<BigDecimal> or BigDecimal[] objects depending on getCardinality().

Deprecated As of 1.1.

105.14.2.2 public static final int BIGINTEGER = 9

The BIGINTEGER type. Attributes of this type should be stored as BigInteger, List<BigInteger> or BigInteger[] objects, depending on the getCardinality() value.

Deprecated As of 1.1.

105.14.2.3 public static final int BOOLEAN = 11

The BOOLEAN type. Attributes of this type should be stored as Boolean, List<Boolean> or boolean[] objects depending on getCardinality().

105.14.2.4 public static final int BYTE = 6

The BYTE type. Attributes of this type should be stored as Byte, List<Byte> or byte[] objects, depending on the getCardinality() value.

105.14.2.5 public static final int CHARACTER = 5

The CHARACTER type. Attributes of this type should be stored as Character, List<Character> or char[] objects, depending on the getCardinality() value.

105.14.2.6 public static final int DOUBLE = 7

The DOUBLE type. Attributes of this type should be stored as Double, List<Double> or double[] objects, depending on the getCardinality() value.

105.14.2.7 public static final int FLOAT = 8

The FLOAT type. Attributes of this type should be stored as Float, List<Float> or float[] objects, depending on the getCardinality() value.

105.14.2.8 public static final int INTEGER = 3

The INTEGER type. Attributes of this type should be stored as Integer, List<Integer> or int[] objects, depending on the getCardinality() value.

105.14.2.9 public static final int LONG = 2

The LONG type. Attributes of this type should be stored as Long, List<Long> or long[] objects, depending on the getCardinality() value.

105.14.2.10 public static final int PASSWORD = 12

The PASSWORD type. Attributes of this type must be stored as String, List<String> or String[] objects depending on getCardinality(). A PASSWORD must be treated as a string but the type can be used to disguise the information when displayed to a user to prevent others from seeing it.

Since 1.2

105.14.2.11 public static final int SHORT = 4

The SHORT type. Attributes of this type should be stored as Short, List<Short> or short[] objects, depending on the getCardinality() value.

105.14.2.12 public static final int STRING = 1

The STRING type.

Attributes of this type should be stored as String, List<String> or String[] objects, depending on the getCardinality() value.

105.14.2.13 public int getCardinality()

□ Return the cardinality of this attribute. The OSGi environment handles multi valued attributes in arrays ([]) or in List objects. The return value is defined as follows:

```
x = Integer.MIN_VALUE    no limit, but use List
x < 0                    -x = max occurrences, store in List
x > 0                     x = max occurrences, store in array []
x = Integer.MAX_VALUE    no limit, but use array []
x = 0                     1 occurrence required
```

Returns The cardinality of this attribute.

105.14.2.14 public String[] getDefaultValue()

□ Return a default for this attribute. The object must be of the appropriate type as defined by the cardinality and getType(). The return type is a list of String objects that can be converted to the appropriate type. The cardinality of the return array must follow the absolute cardinality of this type. For example, if the cardinality = 0, the array must contain 1 element. If the cardinality is 1, it must contain 0 or 1 elements. If it is -5, it must contain from 0 to max 5 elements. Note that the special case of a 0 cardinality, meaning a single value, does not allow arrays or lists of 0 elements.

Returns Return a default value or null if no default exists.

105.14.2.15 public String getDescription()

□ Return a description of this attribute. The description may be localized and must describe the semantics of this type and any constraints.

Returns The localized description of the definition.

105.14.2.16 public String getID()

□ Unique identity for this attribute. Attributes share a global namespace in the registry. For example, an attribute cn or commonName must always be a String and the semantics are always a name of some object. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify an attribute. If such an OID exists, (which can be requested at several standard organizations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a Java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID. It is strongly advised to define the attributes from existing LDAP schemes which will give the OID. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id or oid

105.14.2.17 public String getName()

□ Get the name of the attribute. This name may be localized.

Returns The localized name of the definition.

105.14.2.18 public String[] getOptionLabels()

□ Return a list of labels of option values.

The purpose of this method is to allow menus with localized labels. It is associated with getOptionValues. The labels returned here are ordered in the same way as the values in that method.

If the function returns null, there are no option labels available.

This list must be in the same sequence as the getOptionValues() method. That is, for each index i in getOptionLabels, i in getOptionValues() should be the associated value.

For example, if an attribute can have the value male, female, unknown, this list can return (for dutch) new String[] { "Man", "Vrouw", "Onbekend" }.

Returns A list values

105.14.2.19 public String[] getOptionValues()

□ Return a list of option values that this attribute can take.

If the function returns null, there are no option values available.

Each value must be acceptable to validate() (return "") and must be a String object that can be converted to the data type defined by getType() for this attribute.

This list must be in the same sequence as getOptionLabels(). That is, for each index i in getOptionValues, i in getOptionLabels() should be the label.

For example, if an attribute can have the value male, female, unknown, this list can return new String[] { "male", "female", "unknown" }.

Returns A list values

105.14.2.20 public int getType()

□ Return the type for this attribute.

Defined in the following constants which map to the appropriate Java type. STRING,LONG,INTEGER, SHORT, CHARACTER, BYTE,DOUBLE,FLOAT, BOOLEAN, PASSWORD.

Returns The type for this attribute.

105.14.2.21 public String validate(String value)

value The value before turning it into the basic data type. If the cardinality indicates a multi-valued attribute then the given string must be escaped.

□ Validate an attribute in String form. An attribute might be further constrained in value. This method will attempt to validate the attribute according to these constraints. It can return three different values:

```
null           No validation present
""             No problems detected
"..."          A localized description of why the value is wrong
```

If the cardinality of this attribute is multi-valued then this string must be interpreted as a comma delimited string. The complete value must be trimmed from white space as well as spaces around commas. Commas (',' \u002C) and spaces (' ' \u0020) and backslashes ('\' \u005C) can be escaped with another backslash. Escaped spaces must not be trimmed. For example:

```
value="  a\,b,b\,c,\ c\\,d   " => [ "a,b", "b,c", " c\", "d" ]
```

Returns null, "", or another string

105.14.3 public interface MetaTypeInformation extends MetaTypeProvider

A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

105.14.3.1 public Bundle getBundle()

□ Return the bundle for which this object provides meta type information.

Returns Bundle for which this object provides meta type information.

105.14.3.2 public String[] getFactoryPids()

□ Return the Factory PIDs (for ManagedServiceFactories) for which ObjectClassDefinition information is available.

Returns Array of Factory PIDs.

105.14.3.3 public String[] getPids()

□ Return the PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of PIDs.

105.14.4 public interface MetaTypeProvider

Provides access to metatypes. This interface can be implemented on a Managed Service or Managed Service Factory as well as registered as a service. When registered as a service, it must be registered with a METATYPE_FACTORY_PID or METATYPE_PID service property (or both). Any PID mentioned in either of these factories must be a valid argument to the getObjectClassDefinition(String, String) method.

Concurrency Thread-safe

105.14.4.1 public static final String METATYPE_FACTORY_PID = "metatype.factory.pid"

Service property to signal that this service has ObjectClassDefinition objects for the given factory PIDs. The type of this service property is `String+`.

Since 1.2

105.14.4.2 public static final String METATYPE_PID = "metatype.pid"

Service property to signal that this service has ObjectClassDefinition objects for the given PIDs. The type of this service property is `String+`.

Since 1.2

105.14.4.3 public String[] getLocales()

□ Return a list of available locales. The results must be names that consists of language [_ country [_ variation]] as is customary in the `Locale` class.

Returns An array of locale strings or `null` if there is no locale specific localization can be found.

105.14.4.4 public ObjectClassDefinition getObjectClassDefinition(String id, String locale)

id The ID of the requested object class. This can be a pid or factory pid returned by getPids or getFactoryPids.

locale The locale of the definition or null for default locale.

□ Returns an object class definition for the specified id localized to the specified locale.

The locale parameter must be a name that consists of language["_" country["_" variation]] as is customary in the Locale class. This Locale class is not used because certain profiles do not contain it.

Returns A ObjectClassDefinition object.

Throws IllegalArgumentException– If the id or locale arguments are not valid

105.14.5 public interface MetaTypeService

The MetaType Service can be used to obtain meta type information for a bundle. The MetaType Service will examine the specified bundle for meta type documents to create the returned MetaTypeInformation object.

If the specified bundle does not contain any meta type documents, then a MetaTypeInformation object will be returned that wrappers any ManagedService or ManagedServiceFactory services registered by the specified bundle that implement MetaTypeProvider. Thus the MetaType Service can be used to retrieve meta type information for bundles which contain a meta type documents or which provide their own MetaTypeProvider objects.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

105.14.5.1 public static final String METATYPE_CAPABILITY_NAME = "osgi.metatype"

Capability name for meta type document processors.

Used in Provide-Capability and Require-Capability manifest headers with the osgi.extender namespace. For example:

```
Require-Capability: osgi.extender;
  filter:="(&(osgi.extender=osgi.metatype)(version>=1.4)(!(version>=2.0)))"
```

Since 1.3

105.14.5.2 public static final String METATYPE_DOCUMENTS_LOCATION = "OSGI-INF/metatype"

Location of meta type documents. The MetaType Service will process each entry in the meta type documents directory.

105.14.5.3 public static final String METATYPE_SPECIFICATION_VERSION = "1.4"

Compile time constant for the Specification Version of MetaType Service.

Used in Version and Requirement annotations. The value of this compile time constant will change when the specification version of MetaType Service is updated.

Since 1.4

105.14.5.4 public MetaTypeInformation getMetaTypeInformation(Bundle bundle)

bundle The bundle for which meta type information is requested.

□ Return the MetaType information for the specified bundle.

Returns A MetaTypeInformation object for the specified bundle.

105.14.6 public interface ObjectClassDefinition

Description for the data type information of an objectclass.

Concurrency Thread-safe

105.14.6.1 public static final int ALL = -1

Argument for getAttributeDefinitions(int).

ALL indicates that all the definitions are returned. The value is -1.

105.14.6.2 public static final int OPTIONAL = 2

Argument for getAttributeDefinitions(int).

OPTIONAL indicates that only the optional definitions are returned. The value is 2.

105.14.6.3 public static final int REQUIRED = 1

Argument for getAttributeDefinitions(int).

REQUIRED indicates that only the required definitions are returned. The value is 1.

105.14.6.4 public AttributeDefinition[] getAttributeDefinitions(int filter)

filter ALL,REQUIRED,OPTIONAL

□ Return the attribute definitions for this object class.

Return a set of attributes. The filter parameter can distinguish between ALL,REQUIRED or the OPTIONAL attributes.

Returns An array of attribute definitions or null if no attributes are selected

105.14.6.5 public String getDescription()

□ Return a description of this object class. The description may be localized.

Returns The description of this object class.

105.14.6.6 public InputStream getIcon(int size) throws IOException

size Requested size of an icon. For example, a 16x16 pixel icon has a size of 16

□ Return an InputStream object that can be used to create an icon from.

Indicate the size and return an InputStream object containing an icon. The returned icon maybe larger or smaller than the indicated size.

The icon may depend on the localization.

Returns An InputStream representing an icon or null

Throws IOException– If the InputStream cannot be returned.

105.14.6.7 public String getID()

□ Return the id of this object class.

ObjectDefintion objects share a global namespace in the registry. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organizations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a Java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id of this object class.

105.14.6.8 public String getName()

□ Return the name of this object class. The name may be localized.

Returns The name of this object class.

105.15 org.osgi.service.metatype.annotations

Metatype Annotations Package Version 1.4.

This package is not used at runtime. Annotated classes are processed by tools to generate Meta Type Resources which are used at runtime.

105.15.1 Summary

- `AttributeDefinition` - `AttributeDefinition` information for the annotated method.
- `AttributeType` - Attribute types for the AttributeDefinition annotation.
- `Designate` - Generate a `Designate` element in the Meta Type Resource for an ObjectClassDefinition using the annotated Declarative Services component.
- `Icon` - `Icon` information for an ObjectClassDefinition.
- `ObjectClassDefinition` - Generate a Meta Type Resource using the annotated type.
- `Option` - `Option` information for an AttributeDefinition.
- `RequireMetaTypeExtender` - This annotation can be used to require the Meta Type extender to process metatype resources.
- `RequireMetaTypeImplementation` - This annotation can be used to require the Meta Type implementation.

105.15.2 @AttributeDefinition

`AttributeDefinition` information for the annotated method.

Each method of a type annotated by ObjectClassDefinition has an implied AttributeDefinition annotation. This annotation is only used to specify non-default AttributeDefinition information.

The `id` of this AttributeDefinition is generated from the name of the annotated method as follows:

- A single dollar sign ('`$`' \u0024) is removed unless it is followed by:
  - A low line ('`_`' \u005F) and a dollar sign in which case the three consecutive characters ("`$_$`") are changed to a single hyphen-minus ('`-`' \u002D).
  - Another dollar sign in which case the two consecutive dollar signs ("`$$`") are changed to a single dollar sign.
- A low line ('`_`' \u005F) is changed to a full stop ('`.`' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("`__`") are changed to a single low line.
- All other characters are unchanged.
- If the type declaring the method also declares a `PREFIX_` field whose value is a compile-time constant String, then the id is prefixed with the value of the `PREFIX_` field.

However, if the type annotated by ObjectClassDefinition is a *single-element annotation*, then the id for the `value` method is derived from the name of the annotation type rather than the name of the method. In this case, the simple name of the annotation type, that is, the name of the class without any package name or outer class name, if the annotation type is an inner class, must be converted to the `value` method's id as follows:

- When a lower case character is followed by an upper case character, a full stop ('`.`' \u002E) is inserted between them.
- Each upper case character is converted to lower case.
- All other characters are unchanged.

- If the annotation type declares a PREFIX_ field whose value is a compile-time constant String, then the id is prefixed with the value of the PREFIX_ field.

This id is the value of the id attribute of the generate AD element and is used as the name of the corresponding configuration property.

This annotation is not processed at runtime. It must be processed by tools and used to contribute to a Meta Type Resource document for the bundle.

See Also The AD element of a Meta Type Resource.

Retention CLASS

Target METHOD

105.15.2.1 String name default ""

□ The human readable name of this AttributeDefinition.

If not specified, the name of this AttributeDefinition is derived from the name of the annotated method. For example, low line ('_' \u005F), dollar sign ('$' \u0024), and hyphen-minus ('-' \u002D) are replaced with space (' ' \u0020) and space is inserted between camel case words.

If the name begins with the percent sign ('%' \u0025), the name can be localized.

See Also The name attribute of the AD element of a Meta Type Resource.

105.15.2.2 String description default ""

□ The human readable description of this AttributeDefinition.

If not specified, the description of this AttributeDefinition is the empty string.

If the description begins with the percent sign ('%' \u0025), the description can be localized.

See Also The description attribute of the AD element of a Meta Type Resource.

105.15.2.3 AttributeType type default STRING

□ The type of this AttributeDefinition.

This must be one of the defined attributes types.

If not specified, the type is derived from the return type of the annotated method. Return types of Class and Enum are mapped to STRING. If the return type is List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and
2. have a public no argument constructor,

then the type is derived from the generic type. For example, a return type of List<String> will be mapped to STRING. A return type of a single dimensional array is supported and the type is the component type of the array. Multi dimensional arrays are not supported. Annotation return types are not supported. Any unrecognized type is mapped to STRING. A tool processing the annotation should declare an error for unsupported return types.

See Also The type attribute of the AD element of a Meta Type Resource.

105.15.2.4 int cardinality default 0

□ The cardinality of this AttributeDefinition.

If not specified, the cardinality is derived from the return type of the annotated method. For an array return type, the cardinality is a large positive value. If the return type is List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and

2. have a public no argument constructor,

the cardinality is a large negative value. Otherwise, the cardinality is 0.

See Also The cardinality attribute of the AD element of a Meta Type Resource.

105.15.2.5 String min default ""

□ The minimum value for this AttributeDefinition.

If not specified, there is no minimum value.

See Also The min attribute of the AD element of a Meta Type Resource.

105.15.2.6 String max default ""

□ The maximum value for this AttributeDefinition.

If not specified, there is no maximum value.

See Also The max attribute of the AD element of a Meta Type Resource.

105.15.2.7 String[] defaultValue default {}

□ The default value for this AttributeDefinition.

The specified values are concatenated into a comma delimited list to become the value of the default attribute of the generated AD element.

If not specified and the annotated method is an annotation element that has a default value, then the value of this element is the default value of the annotated element. Otherwise, there is no default value.

See Also The default attribute of the AD element of a Meta Type Resource.

105.15.2.8 boolean required default true

□ The required value for this AttributeDefinition.

If not specified, the value is true.

See Also The required attribute of the AD element of a Meta Type Resource.

105.15.2.9 Option[] options default {}

□ The option information for this AttributeDefinition.

For each specified Option, an Option element is generated for this AttributeDefinition.

If not specified, the option information is derived from the return type of the annotated method. If the return type is an enum, a single dimensional array of an enum, or a List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and
2. have a public no argument constructor,

with a generic type of an enum, then the value of this element has an Option for each value of the enum. The label and value of each Option are set to the name of the corresponding enum value. Otherwise, no Option elements will be generated.

See Also The Option element of a Meta Type Resource.

105.15.3 enum AttributeType

Attribute types for the AttributeDefinition annotation.

See Also AttributeDefinition.type()

105.15.3.1 STRING

The String type.

Attributes of this type should be stored as String, List<String> or String[] objects, depending on the cardinality value.

105.15.3.2 LONG

The Long type.

Attributes of this type should be stored as Long, List<Long> or long[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.3 INTEGER

The Integer type.

Attributes of this type should be stored as Integer, List<Integer> or int[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.4 SHORT

The Short type.

Attributes of this type should be stored as Short, List<Short> or short[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.5 CHARACTER

The Character type.

Attributes of this type should be stored as Character, List<Character> or char[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.6 BYTE

The Byte type.

Attributes of this type should be stored as Byte, List<Byte> or byte[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.7 DOUBLE

The Double type.

Attributes of this type should be stored as Double, List<Double> or double[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.8 FLOAT

The Float type.

Attributes of this type should be stored as Float, List<Float> or float[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.9 BOOLEAN

The Boolean type.

Attributes of this type should be stored as Boolean, List<Boolean> or boolean[] objects depending on AttributeDefinition#cardinality() cardinality.

105.15.3.10 PASSWORD

The Password type.

Attributes of this type must be stored as String, List<String> or String[] objects depending on cardinality.

A Password must be treated as a String but the type can be used to disguise the information when displayed to a user to prevent it from being seen.

105.15.3.11 public String toString()

105.15.3.12 public static AttributeType valueOf(String name)

105.15.3.13 public static AttributeType[] values()

105.15.4 @Designate

Generate a Designate element in the Meta Type Resource for an ObjectClassDefinition using the annotated Declarative Services component.

This annotation must be used on a type that is also annotated with the Declarative Services Component annotation. The component must only have a single PID which is used for the generated Designate element.

This annotation is not processed at runtime. It must be processed by tools and used to contribute to a Meta Type Resource document for the bundle.

See Also The Designate element of a Meta Type Resource.

Retention CLASS

Target TYPE

105.15.4.1 Class<?> ocd

□ The type of the ObjectClassDefinition for this Designate.

The specified type must be annotated with ObjectClassDefinition.

See Also The ocdref attribute of the Designate element of a Meta Type Resource.

105.15.4.2 boolean factory default false

□ Specifies whether this Designate is for a factory PID.

If false, then the PID value from the annotated component will be used in the pid attribute of the generated Designate element. If true, then the PID value from the annotated component will be used in the factoryPid attribute of the generated Designate element.

See Also The pid and factoryPid attributes of the Designate element of a Meta Type Resource.

105.15.5 @Icon

Icon information for an ObjectClassDefinition.

See Also ObjectClassDefinition.icon()

Retention CLASS

Target

105.15.5.1 String resource

□ The resource name for this Icon.

The resource is a URL. The resource URL can be relative to the root of the bundle containing the Meta Type Resource.

If the resource begins with the percent sign ('%' \u0025), the resource can be localized.

See Also The resource attribute of the Icon element of a Meta Type Resource.

105.15.5.2 int size

□ The pixel size of this Icon.

For example, 32 represents a 32x32 icon.

See Also The size attribute of the Icon element of a Meta Type Resource.

105.15.6 @ObjectClassDefinition

Generate a Meta Type Resource using the annotated type.

This annotation can be used without defining any element values since defaults can be generated from the annotated type. Each method of the annotated type has an implied AttributeDefinition annotation if not explicitly annotated.

This annotation may only be used on annotation types and interface types. Use on concrete or abstract class types is unsupported. If applied to an interface then all methods inherited from super types are included as attributes.

This annotation is not processed at runtime. It must be processed by tools and used to generate a Meta Type Resource document for the bundle.

See Also The OCD element of a Meta Type Resource.

Retention CLASS

Target TYPE

105.15.6.1 String id default ""

□ The id of this ObjectClassDefinition.

If not specified, the id of this ObjectClassDefinition is the fully qualified name of the annotated type using the dollar sign ('$' \u0024) to separate nested class names from the name of their enclosing class. The id is not to be confused with a PID which can be specified by the pid() or factoryPid() element.

See Also The id attribute of the OCD element of a Meta Type Resource.

105.15.6.2 String name default ""

□ The human readable name of this ObjectClassDefinition.

If not specified, the name of this ObjectClassDefinition is derived from the id(). For example, low line ('_' \u005F) and dollar sign ('$' \u0024) are replaced with space (' ' \u0020) and space is inserted between camel case words.

If the name begins with the percent sign ('%' \u0025), the name can be localized.

See Also The name attribute of the OCD element of a Meta Type Resource.

105.15.6.3 String description default ""

□ The human readable description of this ObjectClassDefinition.

If not specified, the description of this ObjectClassDefinition is the empty string.

If the description begins with the percent sign ('%' \u0025), the description can be localized.

See Also The description attribute of the OCD element of a Meta Type Resource.

105.15.6.4 String localization default ""

□ The localization resource of this ObjectClassDefinition.

This refers to a resource property entry in the bundle that can be augmented with locale information. If not specified, the localization resource for this ObjectClassDefinition is the string "OSGI-INF/l10n/" followed by the id().

See Also The localization attribute of the MetaData element of a Meta Type Resource.

105.15.6.5 String[] pid default {}

□ The PIDs associated with this ObjectClassDefinition.

For each specified PID, a Designate element with a pid attribute is generated that references this ObjectClassDefinition.

The Designate annotation can also be used to associate a Declarative Services component with an ObjectClassDefinition and generate a Designate element.

A special string ("$") can be used to specify the fully qualified name of the annotated type as a PID. For example:

```
@ObjectClassDefinition(pid="$")
```

Tools creating a Meta Type Resource from this annotation must replace the special string with the fully qualified name of the annotated type.

See Also The pid attribute of the Designate element of a Meta Type Resource., Designate

105.15.6.6 String[] factoryPid default {}

□ The factory PIDs associated with this ObjectClassDefinition.

For each specified factory PID, a Designate element with a factoryPid attribute is generated that references this ObjectClassDefinition.

The Designate annotation can also be used to associate a Declarative Services component with an ObjectClassDefinition and generate a Designate element.

A special string ("$") can be used to specify the fully qualified name of the annotated type as a factory PID. For example:

```
@ObjectClassDefinition(factoryPid="$")
```

Tools creating a Meta Type Resource from this annotation must replace the special string with the fully qualified name of the annotated type.

See Also The factoryPid attribute of the Designate element of a Meta Type Resource., Designate

105.15.6.7 Icon[] icon default {}

□ The icon resources associated with this ObjectClassDefinition.

For each specified Icon, an Icon element is generated for this ObjectClassDefinition. If not specified, no Icon elements will be generated.

See Also The Icon element of a Meta Type Resource.

105.15.7 @Option

Option information for an AttributeDefinition.

See Also AttributeDefinition.options()

Retention CLASS

Target

105.15.7.1 String label default ""

□ The human readable label of this Option.

If not specified, the label of this Option is the empty string.

If the label begins with the percent sign ('%' \u0025), the label can be localized.

See Also The label attribute of the Option element of a Meta Type Resource.

105.15.7.2 String value

□ The value of this Option.

See Also The value attribute of the Option element of a Meta Type Resource.

105.15.8 @RequireMetaTypeExtender

This annotation can be used to require the Meta Type extender to process metatype resources. It can be used directly, or as a meta-annotation.

Since 1.4

Retention CLASS

Target TYPE, PACKAGE

105.15.9 @RequireMetaTypeImplementation

This annotation can be used to require the Meta Type implementation. It can be used directly, or as a meta-annotation.

Since 1.4

Retention CLASS

Target TYPE, PACKAGE

105.16 References

[1] *LDAP.*
https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[2] *Understanding and Deploying LDAP Directory services*
Timothy Howes, et al. ISBN 1-57870-070-1, MacMillan Technical publishing.

[3] *The Java Language Specification, Java SE 8 Edition*
https://docs.oracle.com/javase/specs/jls/se8/html/index.html

106 PreferencesService Specification

Version 1.1

106.1 Introduction

Many bundles need to save some data persistently - in other words, the data is required to survive the stopping and restarting of the bundle and OSGi Framework. In some cases, the data is specific to a particular user. For example, imagine a bundle that implements some kind of game. User specific persistent data could include things like the user's preferred difficulty level for playing the game. Some data is not specific to a user, which we call *system* data. An example would be a table of high scores for the game.

Bundles which need to persist data in an OSGi environment can use the file system via `org.osgi.framework.BundleContext.getDataFile`. A file system, however, can store only bytes and characters, and provides no direct support for named values and different data types.

A popular class used to address this problem for Java applications is the `java.util.Properties` class. This class allows data to be stored as key/value pairs, called *properties*. For example, a property could have a name `com.acme.fudd` and a value of `elmer`. The `Properties` class has rudimentary support for storage and retrieving with its `load` and `store` methods. The `Properties` class, however, has the following limitations:

- Does not support a naming hierarchy.
- Only supports `String` property values.
- Does not allow its content to be easily stored in a back-end system.
- Has no user name-space management.

Since the `Properties` class was introduced in Java 1.0, efforts have been undertaken to replace it with a more sophisticated mechanism. One of these efforts is this Preferences Service specification.

106.1.1 Essentials

The focus of this specification is simplicity, not reliable access to stored data. This specification does *not* define a general database service with transactions and atomicity guarantees. Instead, it is optimized to deliver the stored information when needed, but it will return defaults, instead of throwing an exception, when the back-end store is not available. This approach may reduce the reliability of the data, but it makes the service easier to use, and allows for a variety of compact and efficient implementations.

This API is made easier to use by the fact that many bundles can be written to ignore any problems that the Preferences Service may have in accessing the back-end store, if there is one. These bundles will mostly or exclusively use the methods of the `Preferences` interface which are not declared to throw a `BackingStoreException`.

This service only supports the storage of scalar values and byte arrays. It is not intended for storing large data objects like documents or images. No standard limits are placed on the size of data objects which can be stored, but implementations are expected to be optimized for the handling of small objects.

A hierarchical naming model is supported, in contrast to the flat model of the `Properties` class. A hierarchical model maps naturally to many computing problems. For example, maintaining information about the positions of adjustable seats in a car requires information for each seat. In a hierarchy, this information can be modeled as a node per seat.

A potential benefit of the Preferences Service is that it allows user specific preferences data to be kept in a well defined place, so that a user management system could locate it. This benefit could be useful for such operations as cleaning up files when a user is removed from the system, or to allow a user's preferences to be cloned for a new user.

The Preferences Service does *not* provide a mechanism to allow one bundle to access the preferences data of another. If a bundle wishes to allow another bundle to access its preferences data, it can pass a Preferences or PreferencesService object to that bundle.

The Preferences Service is not intended to provide configuration management functionality. For information regarding Configuration Management, refer to the *Configuration Admin Service Specification* on page 65.

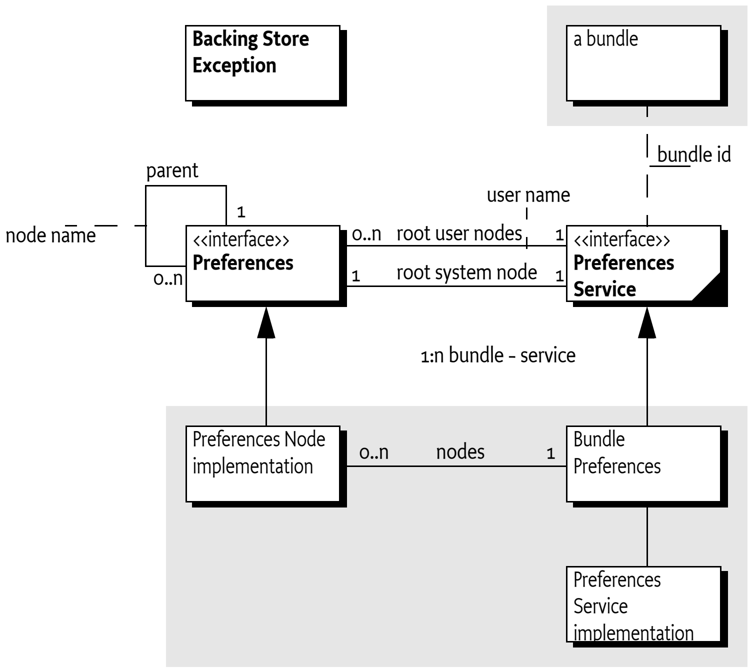
106.1.2 Entities

The PreferencesService is a relatively simple service. It provides access to the different roots of Preferences trees. A single system root node and any number of user root nodes are supported. Each *node* of such a tree is an object that implements the Preferences interface.

This Preferences interface provides methods for traversing the tree, as well as methods for accessing the properties of the node. This interface also contains the methods to flush data into persistent storage, and to synchronize the in-memory data cache with the persistent storage.

All nodes except root nodes have a parent. Nodes can have multiple children.

Figure 106.1 Preferences Class Diagram



106.1.3 Operation

The purpose of the Preferences Service specification is to allow bundles to store and retrieve properties stored in a tree of nodes, where each node implements the Preferences interface. The PreferencesService interface allows a bundle to create or obtain a Preferences tree for system properties, as well as a Preferences tree for each user of the bundle.

This specification allows for implementations where the data is stored locally on the Framework or remotely on a back-end system.

106.2 Preferences Interface

Preferences is an interface that defines the methods to manipulate a node and the tree to which it belongs. A Preferences object contains:

- A set of properties in the form of key/value pairs.
- A parent node.
- A number of child nodes.

106.2.1 Hierarchies

A valid Preferences object always belongs to a *tree*. A tree is identified by its root node. In such a tree, a Preferences object always has a single parent, except for a root node which has a null parent.

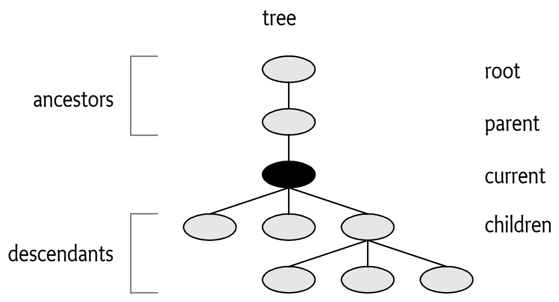
The root node of a tree can be found by recursively calling the parent() method of a node until null is returned. The nodes that are traversed this way are called the *ancestors* of a node.

Each Preferences object has a private name-space for child nodes. Each child node has a name that must be unique among its siblings. Child nodes are created by getting a child node with the node(String) method. The String argument of this call contains a path name. Path names are explained in the next section.

Child nodes can have child nodes recursively. These objects are called the *descendants* of a node.

Descendants are automatically created when they are obtained from a Preferences object, including any intermediate nodes that are necessary for the given path. If this automatic creation is not desired, the nodeExists(String) method can be used to determine if a node already exists.

Figure 106.2 Categorization of nodes in a tree



106.2.2 Naming

Each node has a name relative to its parent. A name may consist of Unicode characters except for the solidus ('/' \u002F). There are no special names, like ".." or ".".

Empty names are reserved for root nodes. Node names that are directly created by a bundle must *always* contain at least one character.

Preferences node names and property keys are *case sensitive*: for example, "org.osgi" and "oRg.oSgI" are two distinct names.

The Preferences Service supports different roots, so there is no absolute root for the Preferences Service. This concept is similar to the Windows Registry that also supports a number of roots.

A path consists of one or more node names, separated by a solidus ('/' \u002F). Paths beginning with a solidus ('/' \u002F) are called *absolute paths* while other paths are called *relative paths*. Paths cannot end with a solidus ('/' \u002F) except for the special case of the root node which has absolute path "/".

Path names are always associated with a specific node; this node is called the current node in the following descriptions. Paths identify nodes as follows.

- *Absolute path* - The first "/" is removed from the path, and the remainder of the path is interpreted as a relative path from the tree's root node.
- *Relative path* -
  - If the path is the empty string, it identifies the current node.
  - If the path is a name (does not contain a "/"), then it identifies the child node with that name.
  - Otherwise, the first name from the path identifies a child of the current node. The name and solidus ('/' \u002F) are then removed from the path, and the remainder of the path is interpreted as a relative path from the child node.

106.2.3 Tree Traversal Methods

A tree can be traversed and modified with the following methods:

- childrenNames() - Returns the names of the child nodes.
- parent() - Returns the parent node.
- removeNode() - Removes this node and all its descendants.
- node(String) - Returns a Preferences object, which is created if it does not already exist. The parameter is an absolute or relative path.
- nodeExists(String) - Returns true if the Preferences object identified by the path parameter exists.

106.2.4 Properties

Each Preferences node has a set of key/value pairs called properties. These properties consist of:

- *Key* - A key is a String object and *case sensitive.*
- The name-space of these keys is separate from that of the child nodes. A Preferences node could have both a child node named fudd and a property named fudd.
- *Value* - A value can always be stored and retrieved as a String object. Therefore, it must be possible to encode/decode all values into/from String objects (though it is not required to store them as such, an implementation is free to store and retrieve the value in any possible way as long as the String semantics are maintained). A number of methods are available to store and retrieve values as primitive types. These methods are provided both for the convenience of the user of the Preferences interface, and to allow an implementation the option of storing the values in a more compact form.

All the keys that are defined in a Preferences object can be obtained with the keys() method. The clear() method can be used to clear all properties from a Preferences object. A single property can be removed with the remove(String) method.

106.2.5 Storing and Retrieving Properties

The Preferences interface has a number of methods for storing and retrieving property values based on their key. All the put* methods take as parameters a key and a value. All the get* methods take as parameters a key and a default value.

- put(String,String), get(String,String)
- putBoolean(String,boolean), getBoolean(String,boolean)
- putInt(String,int), getInt(String,int)
- putLong(String,long), getLong(String,long)
- putFloat(String,float), getFloat(String,float)

- putDouble(String,double), getDouble(String,double)
- putByteArray(String,byte[]), getByteArray(String,byte[])

The methods act as if all the values are stored as String objects, even though implementations may use different representations for the different types. For example, a property can be written as a String object and read back as a float, providing that the string can be parsed as a valid Java float object. In the event of a parsing error, the get* methods do not raise exceptions, but instead return their default parameters.

106.2.6 Defaults

All get* methods take a default value as a parameter. The reasons for having such a default are:

- When a property for a Preferences object has not been set, the default is returned instead. In most cases, the bundle developer does not have to distinguish whether or not a property exists.
- A *best effort* strategy has been a specific design choice for this specification. The bundle developer should not have to react when the back-end store is not available. In those cases, the default value is returned without further notice.

  Bundle developers who want to assure that the back-end store is available should call the flush or sync method. Either of these methods will throw a BackingStoreException if the back-end store is not available.

106.3 Concurrency

This specification specifically allows an implementation to modify Preferences objects in a back-end store. If the back-end store is shared by multiple processes, concurrent updates may cause differences between the back-end store and the in-memory Preferences objects.

Bundle developers can partly control this concurrency with the flush() and sync() method. Both methods operate on a Preferences object.

The flush method performs the following actions:

- Stores (makes persistent) any ancestors (including the current node) that do not exist in the persistent store.
- Stores any properties which have been modified in this node since the last time it was flushed.
- Removes from the persistent store any child nodes that were removed from this object since the last time it was flushed.
- Flushes all existing child nodes.

The sync method will first flush, and then ensure that any changes that have been made to the current node and its descendants in the back-end store (by some other process) take effect. For example, it could fetch all the descendants into a local cache, or it could clear all the descendants from the cache so that they will be read from the back-end store as required.

If either method fails, a BackingStoreException is thrown.

The flush or sync methods provide no atomicity guarantee. When updates to the same back-end store are done concurrently by two different processes, the result may be that changes made by different processes are intermingled. To avoid this problem, implementations may simply provide a dedicated section (or name-space) in the back-end store for each OSGi environment, so that clashes do not arise, in which case there is no reason for bundle programmers to ever call sync.

In cases where sync is used, the bundle programmer needs to take into account that changes from different processes may become intermingled, and the level of granularity that can be assumed is the individual property level. Hence, for example, if two properties need to be kept in lockstep, so that one should not be changed without a corresponding change to the other, consider combining them into a single property, which would then need to be parsed into its two constituent parts.

106.4 PreferencesService Interface

The PreferencesService is obtained from the Framework's service registry in the normal way. Its purpose is to provide access to Preferences root nodes.

A Preferences Service maintains a system root and a number of user roots. User roots are automatically created, if necessary, when they are requested. Roots are maintained on a per bundle basis. For example, a user root called elmer in one bundle is distinct from a user root with the same name in another bundle. Also, each bundle has its own system root. Implementations should use a ServiceFactory service object to create a separate PreferencesService object for each bundle.

The precise description of *user* and *system* will vary from one bundle to another. The Preference Service only provides a mechanism, the bundle may use this mechanism in any desired way.

The PreferencesService interface has the following methods to access the system root and user roots:

- getSystemPreferences() - Return a Preferences object that is the root of the system preferences tree.
- getUserPreferences(String) - Return a Preferences object associated with the user name that is given as argument. If the user does not exist, a new root is created atomically.
- getUsers() - Return an array of the names of all the users for whom a Preferences tree exists.

106.5 Cleanup

The Preferences Service must listen for bundle uninstall events, and remove all the preferences data for the bundle that is being uninstalled. The Preferences Service must use the bundle id for the association and not the location.

It also must handle the possibility of a bundle getting uninstalled while the Preferences Service is stopped. Therefore, it must check on startup whether preferences data exists for any bundle which is not currently installed. If it does, that data must be removed.

106.6 org.osgi.service.prefs

Preferences Service Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.prefs; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.prefs; version="[1.1,1.2)"
```

106.6.1 Summary

- BackingStoreException - Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store.
- Preferences - A node in a hierarchical collection of preference data.
- PreferencesService - The Preferences Service.

106.6.2 public class BackingStoreException extends Exception

Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store.

106.6.2.1 public BackingStoreException(String message)

message The detail message.

□ Constructs a BackingStoreException with the specified detail message.

106.6.2.2 public BackingStoreException(String message, Throwable cause)

message The detail message.

cause The cause of the exception. May be null.

□ Constructs a BackingStoreException with the specified detail message.

Since 1.1

106.6.2.3 public Throwable getCause()

□ Returns the cause of this exception or null if no cause was set.

Returns The cause of this exception or null if no cause was set.

Since 1.1

106.6.2.4 public Throwable initCause(Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws IllegalArgumentException– If the specified cause is this exception.

IllegalStateException– If the cause of this exception has already been set.

Since 1.1

106.6.3 public interface Preferences

A node in a hierarchical collection of preference data.

This interface allows applications to store and retrieve user and system preference data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases.

For each bundle, there is a separate tree of nodes for each user, and one for system preferences. The precise description of "user" and "system" will vary from one bundle to another. Typical information stored in the user preference tree might include font choice, and color choice for a bundle which interacts with the user via a servlet. Typical information stored in the system preference tree might include installation data, or things like high score information for a game program.

Nodes in a preference tree are named in a similar fashion to directories in a hierarchical file system. Every node in a preference tree has a *node name* (which is not necessarily unique), a unique *absolute path name* , and a path name *relative* to each ancestor including itself.

The root node has a node name of the empty String object (""). Every other node has an arbitrary node name, specified at the time it is created. The only restrictions on this name are that it cannot be the empty string, and it cannot contain the slash character ('/').

The root node has an absolute path name of "/". Children of the root node have absolute path names of "/" + *<node name>* . All other nodes have absolute path names of *<parent's absolute path name>* + "/" + *<node name>* . Note that all absolute path names begin with the slash character.

A node *n* 's path name relative to its ancestor *a* is simply the string that must be appended to *a* 's absolute path name in order to form *n* 's absolute path name, with the initial slash character (if present) removed. Note that:

- No relative path names begin with the slash character.
- Every node's path name relative to itself is the empty string.
- Every node's path name relative to its parent is its node name (except for the root node, which does not have a parent).
- Every node's path name relative to the root is its absolute path name with the initial slash character removed.

Note finally that:

- No path name contains multiple consecutive slash characters.
- No path name with the exception of the root's absolute path name end in the slash character.
- Any string that conforms to these two rules is a valid path name.

Each Preference node has zero or more properties associated with it, where a property consists of a name and a value. The bundle writer is free to choose any appropriate names for properties. Their values can be of type String,long,int,boolean, byte[], float, or double but they can always be accessed as if they were String objects.

All node name and property name comparisons are case-sensitive.

All of the methods that modify preference data are permitted to operate asynchronously; they may return immediately, and changes will eventually propagate to the persistent backing store, with an implementation-dependent delay. The flush method may be used to synchronously force updates to the backing store.

Implementations must automatically attempt to flush to the backing store any pending updates for a bundle's preferences when the bundle is stopped or otherwise ungets the Preferences Service.

The methods in this class may be invoked concurrently by multiple threads in a single Java Virtual Machine (JVM) without the need for external synchronization, and the results will be equivalent to some serial execution. If this class is used concurrently *by multiple JVMs* that store their preference data in the same backing store, the data store will not be corrupted, but no other guarantees are made concerning the consistency of the preference data.

No Implement Consumers of this API must not implement this interface

106.6.3.1 public String absolutePath()

□ Returns this node's absolute path name. Note that:

- Root node - The path name of the root node is "/".
- Slash at end - Path names other than that of the root node may not end in slash ('/').
- Unusual names -"." and ".." have *no* special significance in path names.

- Illegal names - The only illegal path names are those that contain multiple consecutive slashes, or that end in slash and are not the root.

Returns this node's absolute path name.

106.6.3.2 public String[] childrenNames() throws BackingStoreException

□ Returns the names of the children of this node. (The returned array will be of size zero if this node has no children and not `null`!)

Returns the names of the children of this node.

Throws `BackingStoreException`– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException`– if this node (or an ancestor) has been removed with the removeNode() method.

106.6.3.3 public void clear() throws BackingStoreException

□ Removes all of the properties (key-value associations) in this node. This call has no effect on any descendants of this node.

Throws `BackingStoreException`– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException`– if this node (or an ancestor) has been removed with the removeNode() method.

See Also remove(String)

106.6.3.4 public void flush() throws BackingStoreException

□ Forces any changes in the contents of this node and its descendants to the persistent store.

Once this method returns successfully, it is safe to assume that all changes made in the subtree rooted at this node prior to the method invocation have become permanent.

Implementations are free to flush changes into the persistent store at any time. They do not need to wait for this method to be called.

When a flush occurs on a newly created node, it is made persistent, as are any ancestors (and descendants) that have yet to be made persistent. Note however that any properties value changes in ancestors are *not* guaranteed to be made persistent.

Throws `BackingStoreException`– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException`– if this node (or an ancestor) has been removed with the removeNode() method.

See Also sync()

106.6.3.5 public String get(String key, String def)

key key whose associated value is to be returned.

def the value to be returned in the event that this node has no value associated with `key` or the backing store is inaccessible.

□ Returns the value associated with the specified `key` in this node. Returns the specified default if there is no value associated with the `key`, or the backing store is inaccessible.

Returns the value associated with `key`, or `def` if no value is associated with `key`.

Throws `IllegalStateException`– if this node (or an ancestor) has been removed with the removeNode() method.

NullPointerException– if key is null. (A null default *is* permitted.)

106.6.3.6 public boolean getBoolean(String key, boolean def)

key key whose associated value is to be returned as a boolean.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a boolean or the backing store is inaccessible.

□ Returns the boolean value represented by the String object associated with the specified key in this node. Valid strings are "true", which represents true, and "false", which represents false. Case is ignored, so, for example, "TRUE" and "False" are also valid. This method is intended for use in conjunction with the putBoolean(String, boolean) method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is something other than "true" or "false", ignoring case.

Returns the boolean value represented by the String object associated with key in this node, or null if the associated value does not exist or cannot be interpreted as a boolean.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also get(String,String), putBoolean(String,boolean)

106.6.3.7 public byte[] getByteArray(String key, byte[] def)

key key whose associated value is to be returned as a byte[] object.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a byte[] type, or the backing store is inaccessible.

□ Returns the byte[] value represented by the String object associated with the specified key in this node. Valid String objects are *Base64* encoded binary data, as defined in RFC 2045 [http://www.ietf.org/rfc/rfc2045.txt], Section 6.8, with one minor change: the string must consist solely of characters from the *Base64 Alphabet*; no newline characters or extraneous characters are permitted. This method is intended for use in conjunction with the putByteArray(String, byte[]) method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is not a valid Base64 encoded byte array (as defined above).

Returns the byte[] value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a byte[].

Throws NullPointerException– if key is null. (A null value for def *is* permitted.)

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also get(String,String), putByteArray(String,byte[])

106.6.3.8 public double getDouble(String key, double def)

key key whose associated value is to be returned as a double value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a double type or the backing store is inaccessible.

□ Returns the double value represented by the String object associated with the specified key in this node. The String object is converted to a double value as by Double.parseDouble(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Double.parseDouble(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putDouble method.

Returns the double value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a double type.

Throws IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

NullPointerException– if key is null.

See Also putDouble(String,double), get(String,String)

106.6.3.9 public float getFloat(String key, float def)

key key whose associated value is to be returned as a float value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a float type or the backing store is inaccessible.

□ Returns the float value represented by the String object associated with the specified key in this node. The String object is converted to a float value as by Float.parseFloat(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Float.parseFloat(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putFloat(String, float) method.

Returns the float value represented by the string associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a float type.

Throws IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

NullPointerException– if key is null.

See Also putFloat(String,float), get(String,String)

106.6.3.10 public int getInt(String key, int def)

key key whose associated value is to be returned as an int .

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as an int or the backing store is inaccessible.

□ Returns the int value represented by the String object associated with the specified key in this node. The String object is converted to an int as by Integer.parseInt(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Integer.parseInt(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putInt(String, int) method.

Returns the int value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as an int type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also putInt(String,int), get(String,String)

106.6.3.11 public long getLong(String key, long def)

key key whose associated value is to be returned as a long value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a long type or the backing store is inaccessible.

□ Returns the long value represented by the String object associated with the specified key in this node. The String object is converted to a long as by Long.parseLong(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if

Long.parseLong(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putLong(String, long) method.

Returns the long value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a long type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also putLong(String,long), get(String,String)

106.6.3.12 public String[] keys() throws BackingStoreException

□ Returns all of the keys that have an associated value in this node. (The returned array will be of size zero if this node has no preferences and not null!)

Returns an array of the keys that have an associated value in this node.

Throws BackingStoreException– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

106.6.3.13 public String name()

□ Returns this node's name, relative to its parent.

Returns this node's name, relative to its parent.

106.6.3.14 public Preferences node(String pathName)

pathName the path name of the Preferences object to return.

□ Returns a named Preferences object (node), creating it and any of its ancestors if they do not already exist. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The empty string ("") is a valid relative pathname, referring to this node itself.

If the returned node did not exist prior to this call, this node and any ancestors that were created by this call are not guaranteed to become persistent until the flush method is called on the returned node (or one of its descendants).

Returns the specified Preferences object.

Throws IllegalArgumentException– if the path name is invalid.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

NullPointerException– if path name is null.

See Also flush()

106.6.3.15 public boolean nodeExists(String pathName) throws BackingStoreException

pathName the path name of the node whose existence is to be checked.

□ Returns true if the named node exists. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The pathname "" is valid, and refers to this node itself.

If this node (or an ancestor) has already been removed with the removeNode() method, it *is* legal to invoke this method, but only with the pathname ""; the invocation will return false. Thus, the idiom p.nodeExists("") may be used to test whether p has been removed.

Returns true if the specified node exists.

Throws BackingStoreException– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method and pathname is not the empty string ("").

IllegalArgumentException– if the path name is invalid (i.e., it contains multiple consecutive slash characters, or ends with a slash character and is more than one character long).

106.6.3.16 public Preferences parent()

□ Returns the parent of this node, or null if this is the root.

Returns the parent of this node.

Throws IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

106.6.3.17 public void put(String key, String value)

key key with which the specified value is to be associated.

value value to be associated with the specified key.

□ Associates the specified value with the specified key in this node.

Throws NullPointerException– if key or value is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

106.6.3.18 public void putBoolean(String key, boolean value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key .

□ Associates a String object representing the specified boolean value with the specified key in this node. The associated string is "true" if the value is true, and "false" if it is false. This method is intended for use in conjunction with the getBoolean(String, boolean) method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports boolean values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a boolean (with getBoolean) or a String (with get) type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getBoolean(String,boolean), get(String,String)

106.6.3.19 public void putByteArray(String key, byte[] value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified byte[] with the specified key in this node. The associated String object the *Base64* encoding of the byte[], as defined in RFC 2045 [http://www.ietf.org/rfc/rfc2045.txt], Section 6.8, with one minor change: the string will consist solely of characters from the *Base64 Alphabet*; it will not contain any newline characters. This method is intended for use in conjunction with the getByteArray(String, byte[]) method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports byte[] values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an a byte[] object (with getByteArray) or a String object (with get).

Throws NullPointerException– if key or value is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getByteArray(String,byte[]), get(String,String)

106.6.3.20 public void putDouble(String key, double value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key .

□ Associates a String object representing the specified double value with the specified key in this node. The associated String object is the one that would be returned if the double value were passed to Double.toString(double). This method is intended for use in conjunction with the getDouble(String, double) method

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports double values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a double (with getDouble) or a String (with get) type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getDouble(String,double)

106.6.3.21 public void putFloat(String key, float value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key .

□ Associates a String object representing the specified float value with the specified key in this node. The associated String object is the one that would be returned if the float value were passed to Float.toString(float). This method is intended for use in conjunction with the getFloat(String, float) method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports float values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a float (with getFloat) or a String (with get) type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getFloat(String,float)

106.6.3.22 public void putInt(String key, int value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified int value with the specified key in this node. The associated string is the one that would be returned if the int value were passed to Integer.toString(int). This method is intended for use in conjunction with getInt(String, int) method.

Implementor's note: it is *not* necessary that the property value be represented by a String object in the backing store. If the backing store supports integer values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an int (with getInt or a String (with get) type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getInt(String,int)

106.6.3.23 public void putLong(String key, long value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified long value with the specified key in this node. The associated String object is the one that would be returned if the long value were passed to Long.toString(long). This method is intended for use in conjunction with the getLong(String, long) method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports long values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a long (with getLong or a String (with get) type.

Throws NullPointerException– if key is null.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also getLong(String,long)

106.6.3.24 public void remove(String key)

key key whose mapping is to be removed from this node.

□ Removes the value associated with the specified key in this node, if any.

Throws IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also get(String,String)

106.6.3.25 public void removeNode() throws BackingStoreException

□ Removes this node and all of its descendants, invalidating any properties contained in the removed nodes. Once a node has been removed, attempting any method other than name(),absolutePath() or nodeExists("") on the corresponding Preferences instance will fail with an IllegalStateException. (The methods defined on Object can still be invoked on a node after it has been removed; they will not throw IllegalStateException.)

The removal is not guaranteed to be persistent until the flush method is called on the parent of this node.

Throws IllegalStateException– if this node (or an ancestor) has already been removed with the removeNode() method.

BackingStoreException– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

See Also flush()

106.6.3.26 public void sync() throws BackingStoreException

□ Ensures that future reads from this node and its descendants reflect any changes that were committed to the persistent store (from any VM) prior to the sync invocation. As a side-effect, forces any changes in the contents of this node and its descendants to the persistent store, as if the flush method had been invoked on this node.

Throws BackingStoreException– if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException– if this node (or an ancestor) has been removed with the removeNode() method.

See Also flush()

106.6.4 public interface PreferencesService

The Preferences Service.

Each bundle using this service has its own set of preference trees: one for system preferences, and one for each user.

A PreferencesService object is specific to the bundle which obtained it from the service registry. If a bundle wishes to allow another bundle to access its preferences, it should pass its PreferencesService object to that bundle.

No Implement Consumers of this API must not implement this interface

106.6.4.1 public Preferences getSystemPreferences()

□ Returns the root system node for the calling bundle.

Returns The root system node for the calling bundle.

106.6.4.2 public Preferences getUserPreferences(String name)

name The user for which to return the preference root node.

□ Returns the root node for the specified user and the calling bundle.

Returns The root node for the specified user and the calling bundle.

106.6.4.3 public String[] getUsers()

□ Returns the names of users for which node trees exist.

Returns The names of users for which node trees exist.

106.7 References

[1] *JSR 10 Preferences API*
https://www.jcp.org/en/jsr/detail?id=10

[2] *RFC 2045 Base 64 encoding*
https://www.ietf.org/rfc/rfc2045.txt

107 User Admin Service Specification

Version 1.1

107.1 Introduction

OSGi frameworks are often used in places where end users or devices initiate actions. These kinds of actions inevitably create a need for authenticating the initiator. Authenticating can be done in many different ways, including with passwords, one-time token cards, biometrics, and certificates.

Once the initiator is authenticated, it is necessary to verify that this principal is authorized to perform the requested action. This authorization can only be decided by the operator of the OSGi environment, and thus requires administration.

The User Admin service provides this type of functionality. Bundles can use the User Admin service to authenticate an initiator and represent this authentication as an `Authorization` object. Bundles that execute actions on behalf of this user can use the `Authorization` object to verify if that user is authorized.

The User Admin service provides authorization based on who runs the code, instead of using the Java code-based permission model. See [1] *The Java Security Architecture for JDK 1.2*. It performs a role similar to [2] *Java Authentication and Authorization Service*.

107.1.1 Essentials

- *Authentication* - A large number of authentication schemes already exist, and more will be developed. The User Admin service must be flexible enough to adapt to the many different authentication schemes that can be run on a computer system.
- *Authorization* - All bundles should use the User Admin service to authenticate users and to find out if those users are authorized. It is therefore paramount that a bundle can find out authorization information with little effort.
- *Security* - Detailed security, based on the Framework security model, is needed to provide safe access to the User Admin service. It should allow limited access to the credentials and other properties.
- *Extensibility* - Other bundles should be able to build on the User Admin service. It should be possible to examine the information from this service and get real-time notifications of changes.
- *Properties* - The User Admin service must maintain a persistent database of users. It must be possible to use this database to hold more information about this user.
- *Administration* - Administering authorizations for each possible action and initiator is time-consuming and error-prone. It is therefore necessary to have mechanisms to group end users and make it simple to assign authorizations to all members of a group at one time.

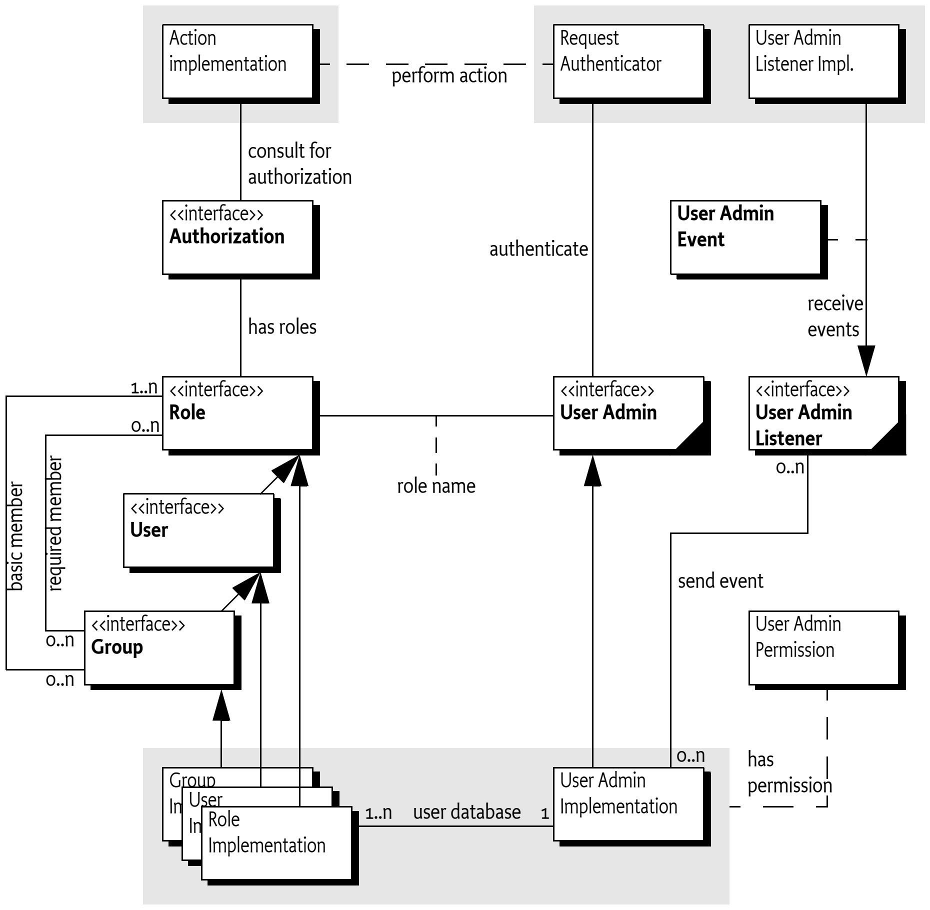
107.1.2 Entities

This Specification defines the following User Admin service entities:

- *User Admin* - This interface manages a database of named roles which can be used for authorization and authentication purposes.
- *Role* - This interface exposes the characteristics shared by all roles: a name, a type, and a set of properties.

- *User* - This interface (which extends Role) is used to represent any entity which may have credentials associated with it. These credentials can be used to authenticate an initiator.
- *Group* - This interface (which extends User) is used to contain an aggregation of named Role objects (Group or User objects).
- *Authorization* - This interface encapsulates an authorization context on which bundles can base authorization decisions.
- *User Admin Event* - This class is used to represent a role change event.
- *User Admin Listener* - This interface provides a listener for events of type UserAdminEvent that can be registered as a service.
- *User Admin Permission* - This permission is needed to configure and access the roles managed by a User Admin service.
- *Role.USER_ANYONE* - This is a special User object that represents *any* user, it implies all other User objects. It is also used when a Group is used with only basic members. The Role.USER_ANYONE is then the only required member.

Figure 107.1 *User Admin Service, org.osgi.service.useradmin*



107.1.3 Operation

An Operator uses the User Admin service to define OSGi framework users and configure them with properties, credentials, and *roles*.

A Role object represents the initiator of a request (human or otherwise). This specification defines two types of roles:

- *User* - A User object can be configured with credentials, such as a password, and properties, such as address, telephone number, and so on.
- *Group* - A Group object is an aggregation of *basic* and *required* roles. Basic and required roles are used in the authorization phase.

An OSGi framework can have several entry points, each of which will be responsible for authenticating incoming requests. An example of an entry point is the Servlet Whiteboard Specification, which delegates authentication of incoming requests to the handleSecurity method of the ServletContextHelper object that was specified when the target servlet or resource of the request was registered.

The OSGi framework entry points should use the information in the User Admin service to authenticate incoming requests, such as a password stored in the private credentials or the use of a certificate.

A bundle can determine if a request for an action is authorized by looking for a Role object that has the name of the requested action.

The bundle may execute the action if the Role object representing the initiator *implies* the Role object representing the requested action.

For example, an initiator Role object *X* implies an action Group object *A* if:

- *X* implies at least one of *A*'s basic members, and
- *X* implies all of *A*'s required members.

An initiator Role object *X* implies an action User object *A* if:

- *A* and *X* are equal.

The Authorization class handles this non-trivial logic. The User Admin service can capture the privileges of an authenticated User object into an Authorization object. The Authorization.hasRole method checks if the authenticate User object has (or implies) a specified action Role object.

For example, in the case of a web application, a server component like a servlet filter can authenticate the initiator and place an Authorization object in the request. The servlet calls the hasRole method on this Authorization object to verify that the initiator has the authority to perform a certain action.

107.2 Authentication

The authentication phase determines if the initiator is actually the one it says it is. Mechanisms to authenticate always need some information related to the user or the OSGi framework to authenticate an external user. This information can consist of the following:

- A secret known only to the initiator.
- Knowledge about cards that can generate a unique token.
- Public information like certificates of trusted signers.
- Information about the user that can be measured in a trusted way.
- Other specific information.

107.2.1 Repository

The User Admin service offers a repository of Role objects. Each Role object has a unique name and a set of properties that are readable by anyone, and are changeable when the changer has the UserAdminPermission. Additionally, User objects, a sub-interface of Role, also have a set of private protected properties called credentials. Credentials are an extra set of properties that are used to authenticate users and that are protected by UserAdminPermission.

Properties are accessed with the Role.getProperties() method and credentials with the User.getCredentials() method. Both methods return a Dictionary object containing key/value pairs. The keys are String objects and the values of the Dictionary object are limited to String or byte[ ] objects.

This specification does not define any standard keys for the properties or credentials. The keys depend on the implementation of the authentication mechanism and are not formally defined by OSGi specifications.

The repository can be searched for objects that have a unique property (key/value pair) with the method UserAdmin.getUser(String,String). This makes it easy to find a specific user related to a specific authentication mechanism. For example, a secure card mechanism that generates unique tokens could have a serial number identifying the user. The owner of the card could be found with the method

```
User owner = useradmin.getUser(
    "secure-card-serial", "132456712-1212" );
```

If multiple User objects have the same property (key *and* value), a null is returned.

There is a convenience method to verify that a user has a credential without actually getting the credential. This is the User.hasCredential(String,Object) method.

Access to credentials is protected on a name basis by UserAdminPermission. Because properties can be read by anyone with access to a User object, UserAdminPermission only protects change access to properties.

107.2.2 Basic Authentication

The following example shows a very simple authentication algorithm based on passwords.

The vendor of the authentication bundle uses the property "com.acme.basic-id" to contain the name of a user as it logs in. This property is used to locate the User object in the repository. Next, the credential "com.acme.password" contains the password and is compared to the entered password. If the password is correct, the User object is returned. In all other cases a SecurityException is thrown.

```
public User authenticate(
        UserAdmin ua, String name, String pwd )
    throws SecurityException {
    User user = ua.getUser("com.acme.basicid",
        username);
    if (user == null)
        throw new SecurityException( "No such user" );

    if (!user.hasCredential("com.acme.password", pwd))
        throw new SecurityException(
            "Invalid password" );
    return user;
}
```

107.2.3 Certificates

Authentication based on certificates does not require a shared secret. Instead, a certificate contains a name, a public key, and the signature of one or more signers.

The name in the certificate can be used to locate a User object in the repository. Locating a User object, however, only identifies the initiator and does not authenticate it.

1. The first step to authenticate the initiator is to verify that it has the private key of the certificate.
2. Next, the User Admin service must verify that it has a User object with the right property, for example "com.acme.certificate"="Fudd".
3. The next step is to see if the certificate is signed by a trusted source. The bundle could use a central list of trusted signers and only accept certificates signed by those sources. Alternatively, it could require that the certificate itself is already stored in the repository under a unique key as a byte[] in the credentials.
4. In any case, once the certificate is verified, the associated User object is authenticated.

107.3 Authorization

The User Admin service authorization architecture is a *role-based model*. In this model, every action that can be performed by a bundle is associated with a *role*. Such a role is a Group object (called group from now on) from the User Admin service repository. For example, if a servlet could be used to activate the alarm system, there should be a group named AlarmSystemActivation.

The operator can administrate authorizations by populating the group with User objects (users) and other groups. Groups are used to minimize the amount of administration required. For example, it is easier to create one Administrators group and add administrative roles to it rather than individually administer all users for each role. Such a group requires only one action to remove or add a user as an administrator.

The authorization decision can now be made in two fundamentally different ways:

An initiator could be allowed to carry out an action (represented by a Group object) if it implied any of the Group object's members. For example, the AlarmSystemActivation Group object contains an Administrators and a Family Group object:

```
    Administrators        = { Elmer, Pepe,Bugs }
    Family                = { Elmer, Pepe, Daffy }

    AlarmSystemActivation = { Administrators, Family}
```

Any of the four members Elmer, Pepe, Daffy, or Bugs can activate the alarm system.

Alternatively, an initiator could be allowed to perform an action (represented by a Group object) if it implied *all* the Group object's members. In this case, using the same AlarmSystemActivation group, only Elmer and Pepe would be authorized to activate the alarm system, since Daffy and Bugs are *not* members of *both* the Administrators and Family Group objects.

The User Admin service supports a combination of both strategies by defining both a set of *basic members* (any) and a set of *required members* (all).

```
Administrators            = { Elmer, Pepe, Bugs}
Family                    = { Elmer, Pepe, Daffy }

AlarmSystemActivation
    required              = { Administrators }
    basic                 = { Family }
```

The difference is made when Role objects are added to the Group object. To add a basic member, use the Group.addMember(Role) method. To add a required member, use the Group.addRequiredMember(Role) method.

Basic members define the set of members that can get access and required members reduce this set by requiring the initiator to *imply* each required member.

A User object implies a Group object if it implies the following:

- *All* of the Group's required members, and
- At *least* one of the Group's basic members

A User object always implies itself.

If only required members are used to qualify the implication, then the standard user Role.USER_ANYONE can be obtained from the User Admin service and added to the Group object. This Role object is implied by anybody and therefore does not affect the required members.

107.3.1 The Authorization Object

The complexity of authorization is hidden in an Authorization class. Normally, the authenticator should retrieve an Authorization object from the User Admin service by passing the authenticated User object as an argument. This Authorization object is then passed to the bundle that performs the action. This bundle checks the authorization with the Authorization.hasRole(String) method. The performing bundle must pass the name of the action as an argument. The Authorization object checks whether the authenticated user implies the Role object, specifically a Group object, with the given name. This is shown in the following example.

```
public void activateAlarm(Authorization auth) {
    if ( auth.hasRole( "AlarmSystemActivation" ) ) {
        // activate the alarm
        ...
    }
    else throw new SecurityException(
        "Not authorized to activate alarm" );
}
```

107.3.2 Authorization Example

This section demonstrates a possible use of the User Admin service. The service has a flexible model and many other schemes are possible.

Assume an Operator installs an OSGi framework. Bundles in this environment have defined the following action groups:

```
AlarmSystemControl
InternetAccess
TemperatureControl
PhotoAlbumEdit
PhotoAlbumView
PortForwarding
```

Installing and uninstalling bundles could potentially extend this set. Therefore, the Operator also defines a number of groups that can be used to contain the different types of system users.

```
Administrators
Buddies
Children
Adults
Residents
```

In a particular instance, the Operator installs it in a household with the following residents and buddies:

```
Residents:      Elmer, Fudd, Marvin, Pepe
Buddies:        Daffy, Foghorn
```

First, the residents and buddies are assigned to the system user groups. Second, the user groups need to be assigned to the action groups.

The following tables show how the groups could be assigned.

Table 107.1 Example Groups with Basic and Required Members

Groups	Elmer	Fudd	Marvin	Pepe	Daffy	Foghorn
Residents	Basic	Basic	Basic	Basic	-	-
Buddies	-	-	-	-	Basic	Basic
Children	-	-	Basic	Basic	-	-
Adults	Basic	Basic	-	-	-	-
Administrators	Basic	-	-	-	-	-

Table 107.2 Example Action Groups with their Basic and Required Members

Groups	Residents	Buddies	Children	Adults	Admin
AlarmSystemControl	Basic	-	-	-	Required
InternetAccess	Basic	-	-	Required	-
TemperatureControl	Basic	-	-	Required	-
PhotoAlbumEdit	Basic	-	Basic	Basic	-
PhotoAlbumView	Basic	Basic	-	-	-
PortForwarding	Basic	-	-	-	Required

107.4 Repository Maintenance

The UserAdmin interface is a straightforward API to maintain a repository of User and Group objects. It contains methods to create new Group and User objects with the createRole(String,int) method. The method is prepared so that the same signature can be used to create new types of roles in the future. The interface also contains a method to remove a Role object.

The existing configuration can be obtained with methods that list all Role objects using a filter argument. This filter, which has the same syntax as the Framework filter, must only return the Role objects for which the filter matches the properties.

Several utility methods simplify getting User objects depending on their properties.

107.5 User Admin Events

Changes in the User Admin service can be determined in real time. Each User Admin service implementation must send a UserAdminEvent object to any service in the Framework service registry that is registered under the UserAdminListener interface. This event must be send asynchronously from the cause of the event. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 8*.

This procedure is demonstrated in the following code sample.

```
class Listener implements UserAdminListener{
    public void roleChanged( UserAdminEvent event ) {
        ...
    }
}
public class MyActivator
    implements BundleActivator {
    public void start( BundleContext context ) {
        context.registerService(
            UserAdminListener.class.getName(),
            new Listener(), null );
    }
    public void stop( BundleContext context ) {}
}
```

It is not necessary to unregister the listener object when the bundle is stopped because the Framework automatically unregisters it. Once registered, the UserAdminListener object must be notified of all changes to the role repository.

107.5.1 Event Admin and User Admin Change Events

User Admin events must be delivered asynchronously to the Event Admin service by the implementation, if present. The topic of a User Admin Event is:

```
org/osgi/service/useradmin/UserAdmin/<eventtype>
```

The following event types are supported:

```
ROLE_CREATED
ROLE_CHANGED
ROLE_REMOVED
```

All User Admin Events must have the following properties:

- event - (UserAdminEvent) The event that was broadcast by the User Admin service.
- role - (Role) The Role object that was created, modified or removed.
- role.name - (String) The name of the role.
- role.type - (Integer) One of ROLE, USER or GROUP.
- service - (ServiceReference) The Service Reference of the User Admin service.
- service.id - (Long) The User Admin service's ID.
- service.objectClass - (String[]) The User Admin service's object class (which must include org.osgi.service.useradmin.UserAdmin)
- service.pid - (String) The User Admin service's persistent identity

107.6 Security

The User Admin service is related to the security model of the OSGi framework, but is complementary to the [1] *The Java Security Architecture for JDK 1.2*. The final permission of most code should be the intersection of the Java 2 Permissions, which are based on the code that is executing, and the User Admin service authorization, which is based on the user for whom the code runs.

107.6.1 User Admin Permission

The User Admin service defines the UserAdminPermission class that can be used to restrict bundles in accessing credentials. This permission class has the following actions:

- changeProperty - This permission is required to modify properties. The name of the permission is the prefix of the property name.
- changeCredential - This action permits changing credentials. The name of the permission is the prefix of the name of the credential.
- getCredential - This action permits getting credentials. The name of the permission is the prefix of the credential.

If the name of the permission is "admin", it allows the owner to administer the repository. No action is associated with the permission in that case.

Otherwise, the permission name is used to match the property name. This name may end with a ".*" string to indicate a wildcard. For example, com.acme.* matches com.acme.fudd.elmer and com.acme.bugs.

107.7 Relation to JAAS

At a glance, the Java Authorization and Authentication Service (JAAS) seems to be a very suitable model for user administration. The OSGi organization, however, decided to develop an independent User Admin service because JAAS was not deemed applicable. The reasons for this include dependency on Java SE version 1.3 ("JDK 1.3") and existing mechanisms in the previous OSGi Service Gateway 1.0 specification.

107.7.1 JDK 1.3 Dependencies

The authorization component of JAAS relies on the java.security.DomainCombiner interface, which provides a means to dynamically update the ProtectionDomain objects affiliated with an AccessControlContext object.

This interface was added in JDK 1.3. In the context of JAAS, the SubjectDomainCombiner object, which implements the DomainCombiner interface, is used to update ProtectionDomain objects. The permissions of ProtectionDomain objects depend on where code came from and who signed it, with permissions based on who is running the code.

Leveraging JAAS would have resulted in user-based access control on the OSGi framework being available only with JDK 1.3, which was not deemed acceptable.

107.7.2 Existing OSGi Mechanism

JAAS provides a plugable authentication architecture, which enables applications and their underlying authentication services to remain independent from each other.

The Servlet Whiteboard Specification already provides a similar feature by allowing servlet and resource registrations to be supported by an ServletContextHelper object, which uses a callback mechanism to perform any required authentication checks before granting access to the servlet or resource. This way, the registering bundle has complete control on a per-servlet and per-resource basis over which authentication protocol to use, how the credentials presented by the remote requester are to be validated, and who should be granted access to the servlet or resource.

107.7.3 Future Road Map

In the future, the main barrier of 1.3 compatibility will be removed. JAAS could then be implemented in an OSGi environment. At that time, the User Admin service will still be needed and will provide complementary services in the following ways:

- The authorization component relies on group membership information to be stored and managed outside JAAS. JAAS does not manage persistent information, so the User Admin service can be a provider of group information when principals are assigned to a Subject object.
- The authorization component allows for credentials to be collected and verified, but a repository is needed to actually validate the credentials.

In the future, the User Admin service can act as the back-end database to JAAS. The only aspect JAAS will remove from the User Admin service is the need for the Authorization interface.

107.8 org.osgi.service.useradmin

User Admin Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.useradmin; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.useradmin; version="[1.1,1.2)"
```

107.8.1 Summary

- Authorization - The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.
- Group - A named grouping of roles (Role objects).
- Role - The base interface for Role objects managed by the User Admin service.
- User - A User role managed by a User Admin service.
- UserAdmin - This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes.
- UserAdminEvent - Role change event.
- UserAdminListener - Listener for UserAdminEvents.
- UserAdminPermission - Permission to configure and access the Role objects managed by a User Admin service.

107.8.2 public interface Authorization

The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.

Bundles associate the privilege to access restricted resources or operations with roles. Before granting access to a restricted resource or operation, a bundle will check if the Authorization object passed to it possess the required role, by calling its hasRole method.

Authorization contexts are instantiated by calling the UserAdmin.getAuthorization(User) method.

Trusting Authorization objects

There are no restrictions regarding the creation of Authorization objects. Hence, a service must only accept Authorization objects from bundles that has been authorized to use the service using code based (or Java 2) permissions.

In some cases it is useful to use ServicePermission to do the code based access control. A service basing user access control on Authorization objects passed to it, will then require that a calling bundle has the ServicePermission to get the service in question. This is the most convenient way. The OSGi environment will do the code based permission check when the calling bundle attempts to get the service from the service registry.

Example: A servlet using a service on a user's behalf. The bundle with the servlet must be given the ServicePermission to get the Http Service.

However, in some cases the code based permission checks need to be more fine-grained. A service might allow all bundles to get it, but require certain code based permissions for some of its methods.

Example: A servlet using a service on a user's behalf, where some service functionality is open to anyone, and some is restricted by code based permissions. When a restricted method is called (e.g., one handing over an Authorization object), the service explicitly checks that the calling bundle has permission to make the call.

No Implement Consumers of this API must not implement this interface

107.8.2.1 public String getName()

□ Gets the name of the User that this Authorization context was created for.

Returns The name of the User object that this Authorization context was created for, or null if no user was specified when this Authorization context was created.

107.8.2.2 public String[] getRoles()

□ Gets the names of all roles implied by this Authorization context.

Returns The names of all roles implied by this Authorization context, or null if no roles are in the context. The predefined role user.anyone will not be included in this list.

107.8.2.3 public boolean hasRole(String name)

name The name of the role to check for.

□ Checks if the role with the specified name is implied by this Authorization context.

Bundles must define globally unique role names that are associated with the privilege of accessing restricted resources or operations. Operators will grant users access to these resources, by creating a Group object for each role and adding User objects to it.

Returns true if this Authorization context implies the specified role, otherwise false.

107.8.3 public interface Group extends User

A named grouping of roles (Role objects).

Whether or not a given Authorization context implies a Group object depends on the members of that Group object.

A Group object can have two kinds of members: *basic* and *required* . A Group object is implied by an Authorization context if all of its required members are implied and at least one of its basic members is implied.

A Group object must contain at least one basic member in order to be implied. In other words, a Group object without any basic member roles is never implied by any Authorization context.

A User object always implies itself.

No loop detection is performed when adding members to Group objects, which means that it is possible to create circular implications. Loop detection is instead done when roles are checked. The semantics is that if a role depends on itself (i.e., there is an implication loop), the role is not implied.

The rule that a Group object must have at least one basic member to be implied is motivated by the following example:

```
 group foo
   required members: marketing
   basic members: alice, bob
```

Privileged operations that require membership in "foo" can be performed only by "alice" and "bob", who are in marketing.

If "alice" and "bob" ever transfer to a different department, anybody in marketing will be able to assume the "foo" role, which certainly must be prevented. Requiring that "foo" (or any Group object for that matter) must have at least one basic member accomplishes that.

However, this would make it impossible for a Group object to be implied by just its required members. An example where this implication might be useful is the following declaration: "Any citizen who is an adult is allowed to vote." An intuitive configuration of "voter" would be:

```
group voter
  required members: citizen, adult
     basic members:
```

However, according to the above rule, the "voter" role could never be assumed by anybody, since it lacks any basic members. In order to address this issue a predefined role named "user.anyone" can be specified, which is always implied. The desired implication of the "voter" group can then be achieved by specifying "user.anyone" as its basic member, as follows:

```
group voter
  required members: citizen, adult
     basic members: user.anyone
```

No Implement Consumers of this API must not implement this interface

107.8.3.1 public boolean addMember(Role role)

role The role to add as a basic member.

□ Adds the specified Role object as a basic member to this Group object.

Returns true if the given role could be added as a basic member, and false if this Group object already contains a Role object whose name matches that of the specified role.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.3.2 public boolean addRequiredMember(Role role)

role The Role object to add as a required member.

□ Adds the specified Role object as a required member to this Group object.

Returns true if the given Role object could be added as a required member, and false if this Group object already contains a Role object whose name matches that of the specified role.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.3.3 public Role[] getMembers()

□ Gets the basic members of this Group object.

Returns The basic members of this Group object, or null if this Group object does not contain any basic members.

107.8.3.4 public Role[] getRequiredMembers()

□ Gets the required members of this Group object.

Returns The required members of this Group object, or null if this Group object does not contain any required members.

107.8.3.5 public boolean removeMember(Role role)

role The Role object to remove from this Group object.

□ Removes the specified Role object from this Group object.

Returns true if the Role object could be removed, otherwise false.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.4 public interface Role

The base interface for Role objects managed by the User Admin service.

This interface exposes the characteristics shared by all Role classes: a name, a type, and a set of properties.

Properties represent public information about the Role object that can be read by anyone. Specific UserAdminPermission objects are required to change a Role object's properties.

Role object properties are Dictionary objects. Changes to these objects are propagated to the User Admin service and made persistent.

Every User Admin service contains a set of predefined Role objects that are always present and cannot be removed. All predefined Role objects are of type ROLE. This version of the org.osgi.service.useradmin package defines a single predefined role named "user.anyone", which is inherited by any other role. Other predefined roles may be added in the future. Since "user.anyone" is a Role object that has properties associated with it that can be read and modified. Access to these properties and their use is application specific and is controlled using UserAdminPermission in the same way that properties for other Role objects are.

No Implement Consumers of this API must not implement this interface

107.8.4.1 public static final int GROUP = 2

The type of a Group role.

The value of GROUP is 2.

107.8.4.2 public static final int ROLE = 0

The type of a predefined role.

The value of ROLE is 0.

107.8.4.3 public static final int USER = 1

The type of a User role.

The value of USER is 1.

107.8.4.4 public static final String USER_ANYONE = "user.anyone"

The name of the predefined role, user.anyone, that all users and groups belong to.

Since 1.1

107.8.4.5 public String getName()

□ Returns the name of this role.

Returns The role's name.

107.8.4.6 public Dictionary<String, Object> getProperties()

□ Returns a Dictionary of the (public) properties of this Role object. Any changes to the returned Dictionary will change the properties of this Role object. This will cause a UserAdminEvent object of type UserAdminEvent.ROLE_CHANGED to be broadcast to any UserAdminListener objects.

Only objects of type String may be used as property keys, and only objects of type String or byte[] may be used as property values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to add, change, or remove a property in the returned Dictionary, a UserAdminPermission named after the property name (or a prefix of it) with action changeProperty is required.

Returns Dictionary containing the properties of this Role object.

107.8.4.7 public int getType()

□ Returns the type of this role.

Returns The role's type.

107.8.5 public interface User extends Role

A User role managed by a User Admin service.

In this context, the term "user" is not limited to just human beings. Instead, it refers to any entity that may have any number of credentials associated with it that it may use to authenticate itself.

In general, User objects are associated with a specific User Admin service (namely the one that created them), and cannot be used with other User Admin services.

A User object may have credentials (and properties, inherited from the Role class) associated with it. Specific UserAdminPermission objects are required to read or change a User object's credentials.

Credentials are Dictionary objects and have semantics that are similar to the properties in the Role class.

No Implement Consumers of this API must not implement this interface

107.8.5.1 public Dictionary<String, Object> getCredentials()

□ Returns a Dictionary of the credentials of this User object. Any changes to the returned Dictionary object will change the credentials of this User object. This will cause a UserAdminEvent object of type UserAdminEvent.ROLE_CHANGED to be broadcast to any UserAdminListeners objects.

Only objects of type String may be used as credential keys, and only objects of type String or of type byte[] may be used as credential values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to retrieve a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action getCredential is required.

In order to add or remove a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action changeCredential is required.

Returns Dictionary object containing the credentials of this User object.

107.8.5.2 public boolean hasCredential(String key, Object value)

key The credential key.

value The credential value.

□ Checks to see if this User object has a credential with the specified key set to the specified value.

If the specified credential value is not of type String or byte[], it is ignored, that is, false is returned (as opposed to an IllegalArgumentException being raised).

Returns true if this user has the specified credential; false otherwise.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission named after the credential key (or a prefix of it) with action getCredential.

107.8.6 public interface UserAdmin

This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes.

This version of the User Admin service defines two types of Role objects: "User" and "Group". Each type of role is represented by an int constant and an interface. The range of positive integers is reserved for new types of roles that may be added in the future. When defining proprietary role types, negative constant values must be used.

Every role has a name and a type.

A User object can be configured with credentials (e.g., a password) and properties (e.g., a street address, phone number, etc.).

A Group object represents an aggregation of User and Group objects. In other words, the members of a Group object are roles themselves.

Every User Admin service manages and maintains its own namespace of Role objects, in which each Role object has a unique name.

No Implement Consumers of this API must not implement this interface

107.8.6.1 public Role createRole(String name, int type)

name The name of the Role object to create.

type The type of the Role object to create. Must be either a Role.USER type or Role.GROUP type.

□ Creates a Role object with the given name and of the given type.

If a Role object was created, a UserAdminEvent object of type UserAdminEvent.ROLE_CREATED is broadcast to any UserAdminListener object.

Returns The newly created Role object, or null if a role with the given name already exists.

Throws IllegalArgumentException– if type is invalid.

SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.6.2 public Authorization getAuthorization(User user)

user The User object to create an Authorization object for, or null for the anonymous user.

□ Creates an Authorization object that encapsulates the specified User object and the Role objects it possesses. The null user is interpreted as the anonymous user. The anonymous user represents a user that has not been authenticated. An Authorization object for an anonymous user will be unnamed, and will only imply groups that user.anyone implies.

Returns the Authorization object for the specified User object.

107.8.6.3 public Role getRole(String name)

name The name of the Role object to get.

□ Gets the Role object with the given name from this User Admin service.

Returns The requested Role object, or null if this User Admin service does not have a Role object with the given name.

107.8.6.4 public Role[] getRoles(String filter) throws InvalidSyntaxException

filter The filter criteria to match.

□ Gets the Role objects managed by this User Admin service that have properties matching the specified LDAP filter criteria. See org.osgi.framework.Filter for a description of the filter syntax. If a null filter is specified, all Role objects managed by this User Admin service are returned.

Returns The Role objects managed by this User Admin service whose properties match the specified filter criteria, or all Role objects if a null filter is specified. If no roles match the filter, null will be returned.

Throws InvalidSyntaxException– If the filter is not well formed.

107.8.6.5 public User getUser(String key, String value)

key The property key to look for.

value The property value to compare with.

□ Gets the user with the given property key-value pair from the User Admin service database. This is a convenience method for retrieving a User object based on a property for which every User object is supposed to have a unique value (within the scope of this User Admin service), such as for example a X.500 distinguished name.

Returns A matching user, if *exactly* one is found. If zero or more than one matching users are found, null is returned.

107.8.6.6 public boolean removeRole(String name)

name The name of the Role object to remove.

□ Removes the Role object with the given name from this User Admin service and all groups it is a member of.

If the Role object was removed, a UserAdminEvent object of type UserAdminEvent.ROLE_REMOVED is broadcast to any UserAdminListener object.

Returns true If a Role object with the given name is present in this User Admin service and could be removed, otherwise false.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.7 public class UserAdminEvent

Role change event.

UserAdminEvent objects are delivered asynchronously to any UserAdminListener objects when a change occurs in any of the Role objects managed by a User Admin service.

A type code is used to identify the event. The following event types are defined: ROLE_CREATED type, ROLE_CHANGED type, and ROLE_REMOVED type. Additional event types may be defined in the future.

See Also UserAdmin, UserAdminListener

107.8.7.1 public static final int ROLE_CHANGED = 2

A Role object has been modified.

The value of ROLE_CHANGED is 0x00000002.

107.8.7.2 public static final int ROLE_CREATED = 1

A Role object has been created.

The value of ROLE_CREATED is 0x00000001.

107.8.7.3 public static final int ROLE_REMOVED = 4

A Role object has been removed.

The value of ROLE_REMOVED is 0x00000004.

107.8.7.4 public UserAdminEvent(ServiceReference<UserAdmin> ref, int type, Role role)

ref The ServiceReference object of the User Admin service that generated this event.

type The event type.

role The Role object on which this event occurred.

□ Constructs a UserAdminEvent object from the given ServiceReference object, event type, and Role object.

107.8.7.5 public Role getRole()

□ Gets the Role object this event was generated for.

Returns The Role object this event was generated for.

107.8.7.6 public ServiceReference<UserAdmin> getServiceReference()

□ Gets the ServiceReference object of the User Admin service that generated this event.

Returns The User Admin service's ServiceReference object.

107.8.7.7 public int getType()

□ Returns the type of this event.

The type values are ROLE_CREATED type, ROLE_CHANGED type, and ROLE_REMOVED type.

Returns The event type.

107.8.8 public interface UserAdminListener

Listener for UserAdminEvents.

UserAdminListener objects are registered with the Framework service registry and notified with a UserAdminEvent object when a Role object has been created, removed, or modified.

UserAdminListener objects can further inspect the received UserAdminEvent object to determine its type, the Role object it occurred on, and the User Admin service that generated it.

See Also UserAdmin, UserAdminEvent

107.8.8.1 public void roleChanged(UserAdminEvent event)

event The UserAdminEvent object.

□ Receives notification that a Role object has been created, removed, or modified.

107.8.9 public final class UserAdminPermission extends BasicPermission

Permission to configure and access the Role objects managed by a User Admin service.

This class represents access to the Role objects managed by a User Admin service and their properties and credentials (in the case of User objects).

The permission name is the name (or name prefix) of a property or credential. The naming convention follows the hierarchical property naming convention. Also, an asterisk may appear at the end of the name, following a ".", or by itself, to signify a wildcard match. For example: "org.osgi.security.protocol.*" or "*" is valid, but "*protocol" or "a*b" are not valid.

The UserAdminPermission with the reserved name "admin" represents the permission required for creating and removing Role objects in the User Admin service, as well as adding and removing members in a Group object. This UserAdminPermission does not have any actions associated with it.

The actions to be granted are passed to the constructor in a string containing a list of one or more comma-separated keywords. The possible keywords are: `changeProperty`,`changeCredential`, and `getCredential`. Their meaning is defined as follows:

```
action
changeProperty    Permission to change (i.e., add and remove)
                  Role object properties whose names start with
                  the name argument specified in the constructor.
changeCredential  Permission to change (i.e., add and remove)
                  User object credentials whose names start
                  with the name argument specified in the constructor.
getCredential     Permission to retrieve and check for the
                  existence of User object credentials whose names
                  start with the name argument specified in the
                  constructor.
```

The action string is converted to lowercase before processing.

Following is a PermissionInfo style policy entry which grants a user administration bundle a number of `UserAdminPermission` object:

```
(org.osgi.service.useradmin.UserAdminPermission "admin")
(org.osgi.service.useradmin.UserAdminPermission "com.foo.*"
              "changeProperty,getCredential,changeCredential")
(org.osgi.service.useradmin.UserAdminPermission "user.*"
                            "changeProperty,changeCredential")
```

The first permission statement grants the bundle the permission to perform any User Admin service operations of type "admin", that is, create and remove roles and configure `Group` objects.

The second permission statement grants the bundle the permission to change any properties as well as get and change any credentials whose names start with `com.foo.`.

The third permission statement grants the bundle the permission to change any properties and credentials whose names start with `user.`. This means that the bundle is allowed to change, but not retrieve any credentials with the given prefix.

The following policy entry empowers the Http Service bundle to perform user authentication:

```
grant codeBase "${jars}http.jar" {
  permission org.osgi.service.useradmin.UserAdminPermission
    "user.password", "getCredential";
};
```

The permission statement grants the Http Service bundle the permission to validate any password credentials (for authentication purposes), but the bundle is not allowed to change any properties or credentials.

Concurrency Thread-safe

107.8.9.1 public static final String ADMIN = "admin"

The permission name "admin".

107.8.9.2 public static final String CHANGE_CREDENTIAL = "changeCredential"

The action string "changeCredential".

107.8.9.3 public static final String CHANGE_PROPERTY = "changeProperty"

The action string "changeProperty".

107.8.9.4 public static final String GET_CREDENTIAL = "getCredential"

The action string "getCredential".

107.8.9.5 public UserAdminPermission(String name, String actions)

name the name of this UserAdminPermission

actions the action string.

□ Creates a new UserAdminPermission with the specified name and actions. name is either the reserved string "admin" or the name of a credential or property, and actions contains a comma-separated list of the actions granted on the specified name. Valid actions are changeProperty,changeCredential, and getCredential.

Throws IllegalArgumentException– If name equals "admin" and actions are specified.

107.8.9.6 public boolean equals(Object obj)

obj the object to be compared for equality with this object.

□ Checks two UserAdminPermission objects for equality. Checks that obj is a UserAdminPermission, and has the same name and actions as this object.

Returns true if obj is a UserAdminPermission object, and has the same name and actions as this UserAdmin-Permission object.

107.8.9.7 public String getActions()

□ Returns the canonical string representation of the actions, separated by comma.

Returns the canonical string representation of the actions.

107.8.9.8 public int hashCode()

□ Returns the hash code value for this object.

Returns A hash code value for this object.

107.8.9.9 public boolean implies(Permission p)

p the permission to check against.

□ Checks if this UserAdminPermission object "implies" the specified permission.

More specifically, this method returns true if:

- *p* is an instanceof UserAdminPermission,
- *p*'s actions are a proper subset of this object's actions, and
- *p*'s name is implied by this object's name. For example, "java.*" implies "java.home".

Returns true if the specified permission is implied by this object; false otherwise.

107.8.9.10 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object for storing UserAdminPermission objects.

Returns a new PermissionCollection object suitable for storing UserAdminPermission objects.

107.8.9.11 public String toString()

□ Returns a string describing this UserAdminPermission object. This string must be in PermissionInfo encoded format.

Returns The PermissionInfo encoded string for this UserAdminPermission object.

See Also org.osgi.service.permissionadmin.PermissionInfo.getEncoded()

107.9 References

[1] *The Java Security Architecture for JDK 1.2*
Version 1.0, Sun Microsystems, October 1998

[2] *Java Authentication and Authorization Service*
https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html

108 Wire Admin Service Specification

Version 1.0

108.1 Introduction

The Wire Admin service is an administrative service that is used to control a wiring topology in the OSGi Framework. It is intended to be used by user interfaces or management programs that control the wiring of services in an OSGi Framework.

The Wire Admin service plays a crucial role in minimizing the amount of context-specific knowledge required by bundles when used in a large array of configurations. The Wire Admin service fulfills this role by dynamically *wiring* services together. Bundles participate in this wiring process by registering services that produce or consume data. The Wire Admin service *wires* the services that produce data to services which consume data.

The purpose of wiring services together is to allow configurable cooperation of bundles in an OSGi Framework. For example, a temperature sensor can be connected to a heating module to provide a controlled system.

The Wire Admin service is a very important OSGi configuration service and is designed to cooperate closely with the Configuration Admin service, as defined in *Configuration Admin Service Specification* on page 65.

108.1.1 Wire Admin Service Essentials

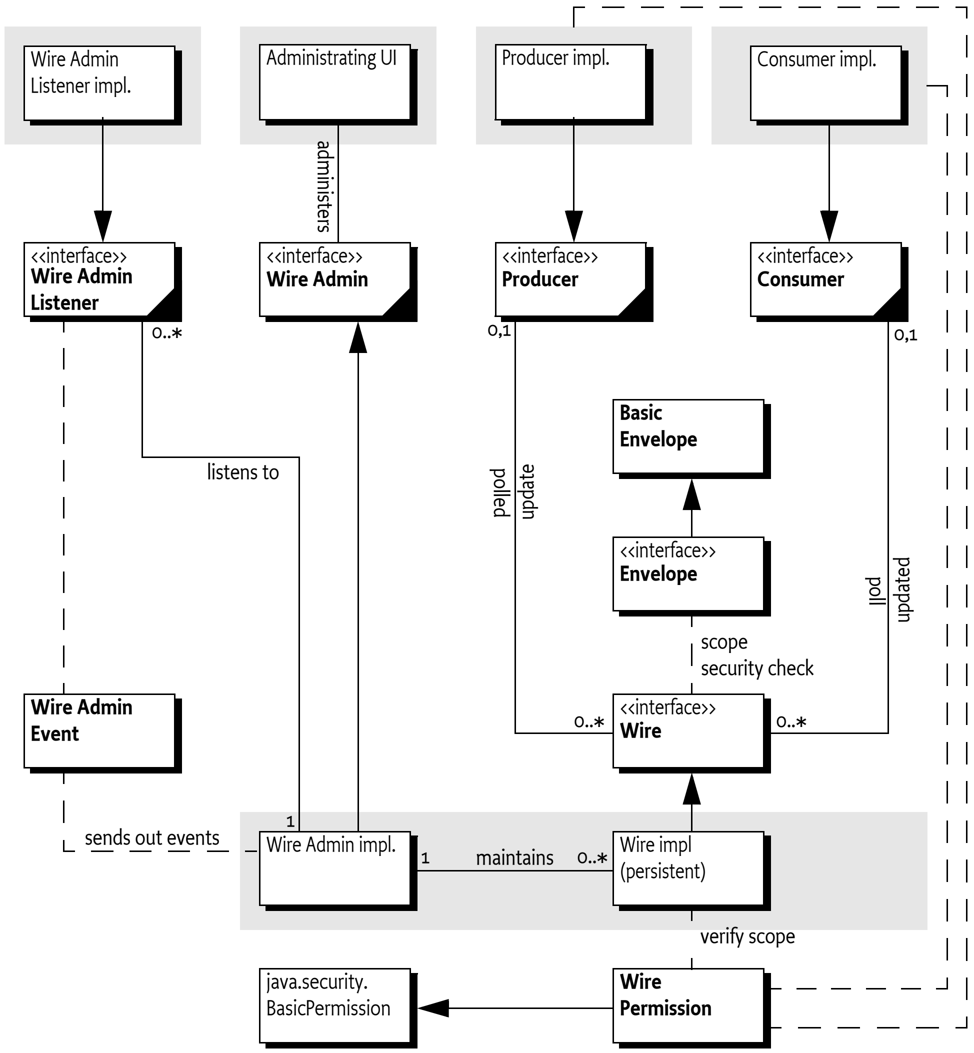
- *Topology Management* - Provide a comprehensive mechanism to link data-producing components with data-consuming components in an OSGi environment.
- *Configuration Management* - Contains configuration data in order to allow either party to adapt to the special needs of the wire.
- *Data Type Handling* - Facilitate the negotiation of the data type to be used for data transfer between producers of data and consumers of data. Consumers and producers must be able to handle multiple data types for data exchanges using a preferred order.
- *Composites* - Support producers and consumers that can handle a large number of data items.
- *Security* - Separate connected parties from each other. Each party must not be required to hold the service object of the other party.
- *Simplicity* - The interfaces should be designed so that both parties, the Producer and the Consumer services, should be easy to implement.

108.1.2 Wire Admin Service Entities

- *Producer* - A service object that generates information to be used by a Consumer service.
- *Consumer* - A service object that receives information generated by a Producer service.
- *Wire* - An object created by the Wire Admin service that defines an association between a Producer service and a Consumer service. Multiple Wire objects can exist between the same Producer and Consumer pair.
- *WireAdmin* - The service that provides methods to create, update, remove, and list Wire objects.
- *WireAdminListener* - A service that receives events from the Wire Admin service when the Wire object is manipulated or used.

- *WireAdminEvent* - The event that is sent to a WireAdminListener object, describing the details of what happened.
- *Configuration Properties* - Properties that are associated with a Wire object and that contain identity and configuration information set by the administrator of the Wire Admin service.
- *PID* - The Persistent IDentity as defined in the Configuration Admin specification.
- *Flavors* - The different data types that can be used to exchange information between Producer and Consumer services.
- *Composite Producer/Consumer* - A Producer/Consumer service that can generate/accept different kinds of values.
- *Envelope* - An interface for objects that can identify a value that is transferred over the wire. Envelope objects contain also a scope name that is used to verify access permissions.
- *Scope* - A set of names that categorizes the kind of values contained in Envelope objects for security and selection purposes.
- *Basic Envelope* - A concrete implementation of the Envelope interface.
- *WirePermission* - A Permission sub-class that is used to verify if a Consumer service or Producer service has permission for specific scope names.
- *Composite Identity* - A name that is agreed between a composite Consumer and Producer service to identify the kind of objects that they can exchange.

Figure 108.1 Class Diagram, org.osgi.service.wireadmin



108.1.3 Operation Summary

The Wire Admin service maintains a set of persistent Wire objects. A Wire object contains a Persistent IDentity (PID) for a Consumer service and a PID for a Producer service. (Wire objects can therefore be created when the Producer or Consumer service is not registered.)

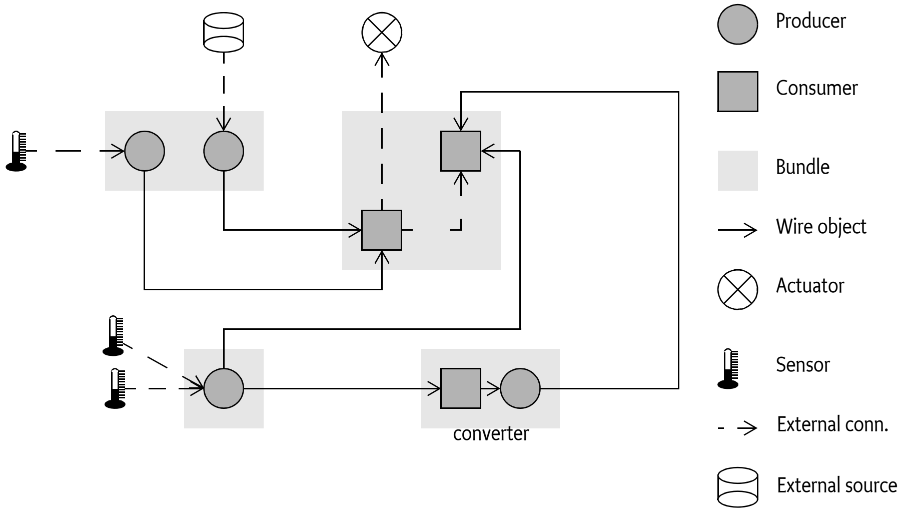
If both those Producer and Consumer services are registered with the Framework, they are connected by the Wire Admin service. The Wire Admin service calls a method on each service object and provides the list of Wire objects to which they are connected.

When a Producer service has new information, it should send this information to each of the connected Wire objects. Each Wire object then must check the filtering and security. If both filtering and security allow the transfer, the Producer service should inform the associated Consumer service with the new information. The Consumer services can also poll a Wire object for an new value at any time.

When a Consumer or Producer service is unregistered from the OSGi Framework, the other object in the association is informed that the Wire object is no longer valid.

Administrative applications can use the Wire Admin service to create and delete wires. These changes are immediately reflected in the current topology and are broadcast to Wire Admin Listener services.

Figure 108.2 An Example Wiring Scheme in an OSGi Environment



108.2 Producer Service

A Producer is a service that can produce a sequence of data objects. For example, a Producer service can produce, among others, the following type of objects:

- Measurement objects that represent a sensor measurement such as temperature, movement, or humidity.
- A String object containing information for user consumption, such as headlines.
- A Date object indicating the occurrence of a periodic event.
- Position information.
- Envelope objects containing status items which can be any type.

108.2.1 Producer Properties

A Producer service must be registered with the OSGi Framework under the interface name org.osgi.service.wireadmin.Producer. The following service properties must be set:

- service.pid - The value of this property, also known as the PID, defines the Persistent IDentity of a service. A Producer service must always use the same PID value whenever it is registered. The PID value allows the Wire Admin service to consistently identify the Producer service and create a persistent Wire object that links a Producer service to a Consumer service. See [1] *Design Patterns* specification for the rules regarding PIDs.
- wireadmin.producer.flavors - The value of this property is an array of Class objects (Class[]) that are the classes of the objects the service can produce. See *Flavors* on page 198 for more information about the data type negotiation between Producer and Consumer services.
- wireadmin.producer.filters - This property indicates to the Wire Admin service that this Producer service performs its own update filtering, meaning that the consumer can limit the number of update calls with a filter expression. This does not modify the data; it only determines whether an update via the wire occurs. If this property is not set, the Wire object must filter according to

the description in *Composite objects* on page 192. This service registration property does not need to have a specific value.

- wireadmin.producer.scope - Only for a composite Producer service, a list of scope names that define the scope of this Producer service, as explained in *Scope* on page 193.
- wireadmin.producer.composite - List the composite identities of Consumer services with which this Producer service can interoperate. This property is of type String[]. A composite Consumer service can inter-operate with a composite Producer service when there is at least one name that occurs in both the Consumer service's array and the Producer service's array for this property.

108.2.2 Connections

The Wire Admin service connects a Producer service and a Consumer service by creating a Wire object. If the Consumer and Producer services that are bound to a Wire object are registered with the Framework, the Wire Admin service must call the consumersConnected(Wire[]) method on the Producer service object. Every change in the Wire Admin service that affects the Wire object to which a Producer service is connected must result in a call to this method. This requirement ensures that the Producer object is informed of its role in the wiring topology. If the Producer service has no Wire objects attached when it is registered, the Wire Admin service must always call consumersConnected(null). This situation implies that a Producer service can assume it always gets called back from the Wire Admin service when it registers.

108.2.3 Producer Example

The following example shows a clock producer service that sends out a Date object every second.

```
public class Clock extends Thread implementsProducer {
    Wire          wires[];
    BundleContext context;
    boolean       quit;

    Clock( BundleContext context ) {
        this.context = context;
        start();
    }
    public synchronized void run() {
        Hashtable p = new Hashtable();
        p.put( org.osgi.service.wireadmin.WireConstants.
                        WIREADMIN_PRODUCER_FLAVORS,
             new Class[] { Date.class } );
        p.put( org.osgi.framework.Constants.SERVICE_PID,
            "com.acme.clock" );
        context.registerService(
            Producer.class.getName(),this,p );

        while( ! quit )
        try {
            Date now = new Date();
            for( int i=0; wires!=null && i<wires.length;i++ )
                wires[i].update( now );
            wait( 1000 );
        }
        catch( InterruptedException ie) {
            /* will recheck quit */
        }
    }
    public void synchronized consumersConnected(Wire wires[])
    {
        this.wires = wires;
    }
    public Object polled(Wire wire) { return new Date(); }
  ...
}
```

108.2.4 Push and Pull

Communication between Consumer and Producer services can be initiated in one of the following ways.

- The Producer service calls the update(Object) method on the Wire object. The Wire object implementation must then call the updated(Wire,Object) method on the Consumer service, if the filtering allows this.
- The Consumer service can call poll() on the Wire object. The Wire object must then call polled(Wire) on the Producer object. Update filtering must not apply to polling.

108.2.5 Producers and Flavors

Consumer services can only understand specific data types, and are therefore restricted in what data they can process. The acceptable object classes, the flavors, are communicated by the Consumer service to the Wire Admin service using the Consumer service's service registration properties. The method getFlavors() on the Wire object returns this list of classes. This list is an ordered list in which the first class is the data type that is the most preferred data type supported by the Consumer service. The last class is the least preferred data type. The Producer service must attempt to convert its data into one of the data types according to the preferred order, or will return null from the poll method to the Consumer service if none of the types are recognized.

Classes cannot be easily compared for equivalence. Sub-classes and interfaces allow classes to masquerade as other classes. The Class.isAssignableFrom(Class) method verifies whether a class is type compatible, as in the following example:

```
Object polled(Wire wire) {
    Class clazzes[] = wire.getFlavors();
    for ( int i=0; i<clazzes.length; i++ ) {
        Class clazz = clazzes[i];
        if ( clazz.isAssignableFrom( Date.class ) )
            return new Date();
        if (    clazz.isAssignableFrom( String.class) )
            return new Date().toString();
    }
    return null;
}
```

The order of the if statements defines the preferences of the Producer object. Preferred data types are checked first. This order normally works as expected but in rare cases, sub-classes can change it. Normally, however, that is not a problem.

108.3 Consumer Service

A Consumer service is a service that receives information from one or more Producer services and is wired to Producer services by the Wire Admin service. Typical Consumer services are as follows:

- The control of an actuator, such as a heating element, oven, or electric shades
- A display

- A log
- A state controller such as an alarm system

108.3.1 Consumer Properties

A Consumer service must be registered with the OSGi Framework under the interface name org.osgi.service.wireadmin.Consumer. The following service properties must be set:

- service.pid - The value of this property, also known as the PID, defines the Persistent IDentity of a service. A Consumer service must always use the same PID value whenever it is registered. The PID value allows the Wire Admin service to consistently identify the Consumer service and create a persistent Wire object that links a Producer service to a Consumer service. See the Configuration Admin specification for the rules regarding PIDs.
- wireadmin.consumer.flavors - The value of this property is an array of Class objects (Class[]) that are the acceptable classes of the objects the service can process. See *Flavors* on page 198 for more information about the data type negotiation between Producer and Consumer services.
- wireadmin.consumer.scope - Only for a composite Consumer service, a list of scope names that define the scope of this Consumer service, as explained in *Scope* on page 193.
- wireadmin.consumer.composite - List the composite identities of Producer services that this Consumer service can interoperate with. This property is of type String[]. A composite Consumer service can interoperate with a composite Producer service when at least one name occurs in both the Consumer service's array and the Producer service's array for this property.

108.3.2 Connections

When a Consumer service is registered and a Wire object exists that associates it to a registered Producer service, the producersConnected(Wire[]) method is called on the Consumer service.

Every change in the Wire Admin service that affects a Wire object to which a Consumer service is connected must result in a call to the producersConnected(Wire[]) method. This rule ensures that the Consumer object is informed of its role in the wiring topology. If the Consumer service has no Wire objects attached, the argument to the producersConnected(Wire[]) method must be null. This method must also be called when a Producer service registers for the first time and no Wire objects are available.

108.3.3 Consumer Example

For example, a service can implement a Consumer service that logs all objects that are sent to it in order to allow debugging of a wiring topology.

```
public class LogConsumer implements Consumer{
    public LogConsumer( BundleContext context ) {
        Hashtable ht = new Hashtable();
        ht.put(
            Constants.SERVICE_PID, "com.acme.logconsumer" );
        ht.put( WireConstants.WIREADMIN_CONSUMER_FLAVORS,
            new Class[] { Object.class } );
        context.registerService( Consumer.class.getName(),
            this, ht );
    }
    public void updated( Wire wire, Object o ) {
        getLog().log( LogService.LOG_INFO, o.toString() );
    }
    public void producersConnected( Wire [] wires) {}
    LogService getLog() { ... }
}
```

108.3.4 Polling or Receiving a Value

When the Producer service produces a new value, it calls the update(Object) method on the Wire object, which in turn calls the updated(Wire,Object) method on the Consumer service object. When the Consumer service needs a value immediately, it can call the poll() method on the Wire object which in turn calls the polled(Wire) method on the Producer service.

If the poll() method on the Wire object is called and the Producer is unregistered, it must return a null value.

108.3.5 Consumers and Flavors

Producer objects send objects of different data types through Wire objects. A Consumer service object should offer a list of preferred data types (classes) in its service registration properties. The Producer service, however, can still send a null object or an object that is not of the preferred types. Therefore, the Consumer service must check the data type and take the appropriate action. If an object type is incompatible, then a log message should be logged to allow the operator to correct the situation.

The following example illustrates how a Consumer service can handle objects of type Date, Measurement, and String.

```
void process( Object in ) {
    if ( in instanceof Date )
        processDate( (Date) in );
    else if ( in instanceof Measurement )
        processMeasurement( (Measurement) in );
    else if ( in instanceof String )
        processString( (String) in );
    else
        processError( in );
}
```

108.4 Implementation issues

The Wire Admin service can call the consumersConnected or producersConnected methods during the registration of the Consumer or Producer service. Care should be taken in this method call so that no variables are used that are not yet set, such as the ServiceRegistration object that is returned from the registration. The same is true for the updated or polled callback because setting the Wire objects on the Producer service causes such a callback from the consumersConnected or producersConnected method.

A Wire Admin service must call the producersConnected and consumersConnected method asynchronously from the registrations, meaning that the Consumer or Producer service can use synchronized to restrict access to critical variables.

When the Wire Admin service is stopped, it must disconnect all connected consumers and producers by calling producersConnected and consumersConnected with a null for the wires parameter.

108.5 Wire Properties

A Wire object has a set of properties (a Dictionary object) that configure the association between a Consumer service and a Producer service. The type and usage of the keys, as well as the allowed types for the values are defined in *Configuration Properties* on page 72.

The Wire properties are explained in the following table.

Table 108.1 Standard Wire Properties

Constant	Description
WIREADMIN_PID	The value of this property is a unique Persistent IDentity as defined in chapter *Configuration Admin Service Specification* on page 65. This PID must be automatically created by the Wire Admin service for each new Wire object.
WIREADMIN_PRODUCER_PID	The value of the property is the PID of the Producer service.
WIREADMIN_CONSUMER_PID	The value of this property is the PID of the Consumer service.
WIREADMIN_FILTER	The value of this property is an OSGi filter string that is used to control the update of produced values. This filter can contain a number of attributes as explained in *Wire Flow Control* on page 195.

The properties associated with a Wire object are not limited to the ones defined in Table 108.1. The Dictionary object can also be used for configuring *both* Consumer services and Producer services. Both services receive the Wire object and can inspect the properties and adapt their behavior accordingly.

108.5.1 Display Service Example

In the following example, the properties of a Wire object, which are set by the Operator or User, are used to configure a Producer service that monitors a user's email account regularly and sends a message when the user has received email. This WireMail service is illustrated as follows:

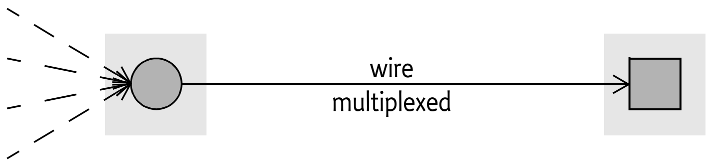
```
public class WireMail extends Thread
    implements Producer {
    Wire          wires[];
    BundleContext  context;
    boolean        quit;

    public void start( BundleContext context ) {
        Hashtable ht = new Hashtable();
        ht.put( Constants.SERVICE_PID, "com.acme.wiremail" );
        ht.put( WireConstants.WIREADMIN_PRODUCER_FLAVORS,
             new Class[] { Integer.class } );
        context.registerService( this,
            Producer.class.getName(),
            ht );
    }
    public synchronized void  consumersConnected(
        Wire wires[] ) {
        this.wires = wires;
    }
    public Object polled( Wire wire  ) {
        Dictionary p = wire.getProperties();
        // The password should be
        // obtained from User Admin Service
        int n = getNrMails(
            p.get( "userid" ),
            p.get( "mailhost" ) );
        return new Integer( n );
    }
    public synchronized void run() {
        while ( !quit )
        try {
            for ( int i=0; wires != null && i<wires.length;i++)
                wires[i].update( polled( wires[i] ) );

            wait( 150000 );
        }
        catch( InterruptedException e ) { break; }
    }
    ...
}
```

108.6 Composite objects

A Producer and/or Consumer service for each information item is usually the best solution. This solution is not feasible, however, when there are hundreds or thousands of information items. Each registered Consumer or Producer service carries the overhead of the registration, which may overwhelm a Framework implementation on smaller platforms.

When the size of the platform is an issue, a Producer and a Consumer service should abstract a larger number of information items. These Consumer and Producer services are called *composite*.

Figure 108.3 *Composite Producer Example*



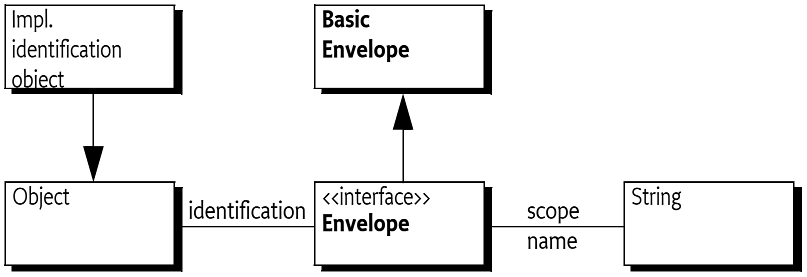
Composite Producer and Consumer services should register respectively the WIREADMIN_PRODUCER_COMPOSITE and WIREADMIN_CONSUMER_COMPOSITE *composite identity* property with their service registration. These properties should contain a list of composite identities. These identities are not defined here, but are up to a mutual agreement between the Consumer and Producer service. For example, a composite identity could be MOST-1.5 or GSM-Phase2-Terminal. The name may follow any scheme but will usually have some version information embedded. The composite identity properties are used to match Consumer and Producer services with each other during configuration of the Wire Admin service. A Consumer and Producer service should inter-operate when at least one equal composite identity is listed in both the Producer and Consumer composite identity service property.

Composite producers/consumers must identify the *kind* of objects that are transferred over the Wire object, where *kind* refers to the intent of the object, not the data type. For example, a Producer service can represent the status of a door-lock and the status of a window as a boolean. If the status of the window is transferred as a boolean to the Consumer service, how would it know that this boolean represents the window and not the door-lock

To avoid this confusion, the Wire Admin service includes an Envelope interface. The purpose of the Envelope interface is to associate a value object with:

- An identification object
- A scope name

Figure 108.4 Envelope



108.6.1 Identification

The Envelope object's identification object is used to identify the value carried in the Envelope object. Each unique kind of value must have its own unique identification object. For example, a left-front-window should have a different identification object than a rear-window.

The identification is of type Object. Using the Object class allows String objects to be used, but also makes it possible to use more complex objects. These objects can convey information in a way that is mutually agreed between the Producer and Consumer service. For example, its type may differ depending on each kind of value so that the *Visitor* pattern, see [1] *Design Patterns*, can be used. Or it may contain specific information that makes the Envelope object easier to dispatch for the Consumer service.

108.6.2 Scope

The scope name is a String object that *categorizes* the Envelope object. The scope name is used to limit the kind of objects that can be exchanged between composite Producer and Consumer services, depending on security settings.

The name-space for this scope should be mutually agreed between the Consumer and Producer services a priori. For the Wire Admin service, the scope name is an opaque string. Its syntax is specified in *Scope name syntax* on page 195.

Both composite Producer and Consumer services must add a list of their supported scope names to the service registration properties. This list is called the *scope* of that service. A Consumer service must add this scope property with the name of WIREADMIN_CONSUMER_SCOPE, a Producer service must add this scope property with the name WIREADMIN_PRODUCER_SCOPE. The type of this property must be a String[] object.

Not registering this property by the Consumer or the Producer service indicates to the Wire Admin service that any Wire object connected to that service must return null for the Wire.getScope() method. This case must be interpreted by the Consumer or Producer service that no scope verification is taking place. Secure Producer services should not produce values for this Wire object and secure Consumer services should not accept values.

It is also allowed to register with a *wildcard*, indicating that all scope names are supported. In that case, the WIREADMIN_SCOPE_ALL (which is String[] { "*" }) should be registered as the scope of the service. The Wire object's scope is then fully defined by the other service connected to the Wire object.

The following example shows how a scope is registered.

```
static String [] scope = { "DoorLock", "DoorOpen","VIN" };

public void start( BundleContext context ) {
    Dictionary properties = new Hashtable();
    properties.put(
        WireConstants.WIREADMIN_CONSUMER_SCOPE,
        scope );
    properties.put( WireConstants.WIREADMIN_CONSUMER_PID,
        "com.acme.composite.consumer" );
    properties.put(
        WireConstants.WIREADMIN_CONSUMER_COMPOSITE,
        new String[] { "OSGiSP-R3" } );
    context.registerService( Consumer.class.getName(),
        new AcmeConsumer(),
        properties );
}
```

Both a composite Consumer and Producer service must register a scope to receive scope support from the Wire object. These two scopes must be converted into a single Wire object's scope and scope names in this list must be checked for the appropriate permissions. This resulting scope is available from the Wire.getScope() method.

If no scope is set by either the Producer or the Consumer service the result must be null. In that case, the Producer or Consumer service must assume that no security checking is in place. A secure Consumer or Producer service should then refuse to operate with that Wire object.

Otherwise, the resulting scope is the intersection of the Consumer and Producer service scope where each name in the scope, called m, must be implied by a WirePermission[m,CONSUME] of the Consumer service, and WirePermission[m,PRODUCE] of the Producer service.

If either the Producer or Consumer service has registered a wildcard scope then it must not restrict the list of the other service, except for the permission check. If both the Producer and Consumer service registered a wild-card, the resulting list must be WIREADMIN_SCOPE_ALL (String[]{"*"}).

For example, the Consumer service has registered a scope of {A,B,C} and has WirePermission[*,CONSUME]. The Producer service has registered a scope of {B,C,E} and has WirePermission[C|E, PRODUCE,]. The resulting scope is then {C}. The following table shows this and more examples.

Table 108.2 Examples of scope calculation. C=Consumer, P=Producer, p=WirePermission, s=scope

Cs	Cp	Ps	Pp	Wire Scope
null		null		null
{A,B,C}	*	null		null
null		{C,D,E}		null
{A,B,C}	B|C	{A,B,C}	A|B	{B}
*	*	{A,B,C}	A|B|C	{A,B,C}
*	*	*	*	{*}
{A,B,C}	A|B|C	{A,B,C}	X	{}
{A,B,C}	*	{B,C,E}	C|E	{C}

The Wire object's scope must be calculated only once, when both the Producer and Consumer service become connected. When a Producer or Consumer service subsequently modifies its scope, the Wire object must *not* modify the original scope. A Consumer and a Produce service can thus assume that the scope does not change after the producersConnected method or consumersConnected method has been called.

108.6.3 Access Control

When an Envelope object is used as argument in Wire.update(Object) then the Wire object must verify that the Envelope object's scope name is included in the Wire object's scope. If this is not the case, the update must be ignored (the updated method on the Consumer service must not be called).

A composite Producer represents a number of values, which is different from a normal Producer that can always return a single object from the poll method. A composite Producer must therefore return an array of Envelope objects (Envelope[]). This array must contain Envelope objects for all the values that are in the Wire object's scope. It is permitted to return all possible values for the Producer because the Wire object must remove all Envelope objects that have a scope name not listed in the Wire object's scope.

108.6.4 Composites and Flavors

Composite Producer and Consumer services must always use a flavor of the Envelope class. The data types of the values must be associated with the scope name or identification and mutually agreed between the Consumer and Producer services.

Flavors and Envelope objects both represent categories of different values. Flavors, however, are different Java classes that represent the same kind of value. For example, the tire pressure of the left front wheel could be passed as a Float, an Integer, or a Measurement object. Whatever data type is chosen, it is still the tire pressure of the left front wheel. The Envelope object represents the kind of object, for example the right front wheel tire pressure, or the left rear wheel.

108.6.5 Scope name syntax

Scope names are normal String objects and can, in principle, contain any Unicode character. In use, scope names can be a full wildcard ('*') but they cannot be partially wildcarded for matching scopes.

Scope names are used with the WirePermission class that extends java.security.BasicPermission. The BasicPermission class implements the implies method and performs the name matching. The wildcard matching of this class is based on the concept of names where the constituents of the name are separated with a period ('.'): for example, org.osgi.service.http.port.

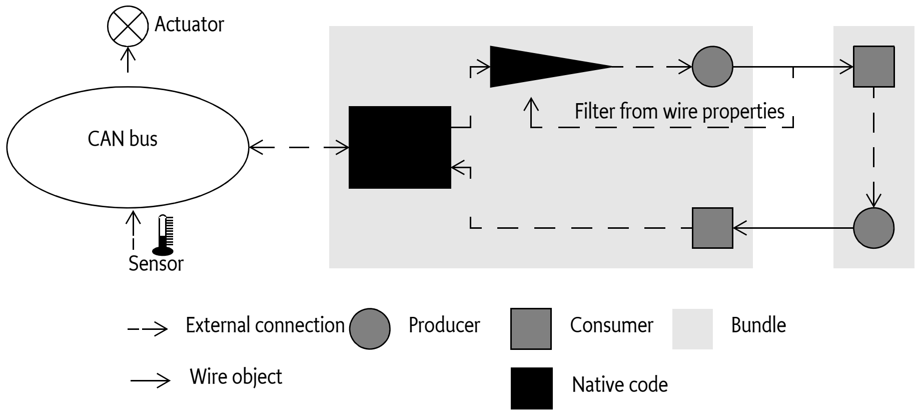
Scope names must therefore follow the rules for fully qualified Java class names. For example, door.lock is a correct scope name while door-lock is not.

108.7 Wire Flow Control

The WIREADMIN_FILTER property contains a filter expression (as defined in the OSGi Framework Filter class) that is used to limit the number of updates to the Consumer service. This is necessary because information can arrive at a much greater rate than can be processed by a Consumer service. For example, a single CAN bus (the electronic control bus used in current cars) in a car can easily deliver hundreds of measurements per second to an OSGi based controller. Most of these measurements are not relevant to the OSGi bundles, at least not all the time. For example, a bundle that maintains an indicator for the presence of frost is only interested in measurements when the outside temperature passes the 4 degrees Celsius mark.

Limiting the number of updates from a Producer service can make a significant difference in performance (meaning that less hardware is needed). For example, a vendor can implement the filter in native code and remove unnecessary updates prior to processing in the Java Virtual Machine (JVM). This is depicted in Figure 108.5 on page 196.

Figure 108.5 *Filtering of Updates*



The filter can use any combination of the following attributes in a filter to implement many common filtering schemes:

Table 108.3 *Filter Attribute Names*

Constant	Description
WIREVALUE_CURRENT	Current value of the data from the Producer service.
WIREVALUE_PREVIOUS	Previous data value that was reported to the Consumer service.
WIREVALUE_DELTA_ABSOLUTE	The actual positive difference between the previous data value and the current data value. For example, if the previous data value was 3 and the current data value is -0.5, then the absolute delta is 3.5. This filter attribute is not set when the current or previous value is not a number.
WIREVALUE_DELTA_RELATIVE	The absolute (meaning always positive) relative change between the current and the previous data values, calculated with the following formula: |previous-current|/|current|. For example, if the previous value was 3 and the new value is 5, then the relative delta is |3-5|/|5| = 0.4. This filter attribute is not set when the current or previous value is not a number.
WIREVALUE_ELAPSED	The time in milliseconds between the last time the Consumer. updated(Wire,Object) returned and the time the filter is evaluated.

Filter attributes can be used to implement many common filtering schemes that limit the number of updates that are sent to a Consumer service. The Wire Admin service specification requires that updates to a Consumer service are always filtered if the WIREADMIN_FILTER Wire property is present. Producer services that wish to perform the filtering themselves should register with a service property WIREADMIN_PRODUCER_FILTERS. Filtering must be performed by the Wire object for all other Producer services.

Filtering for composite Producer services is not supported. When a filter is set on a Wire object, the Wire must still perform the filtering (which is limited to time filtering because an Envelope object is not a magnitude), but this approach may lose relevant information because the objects are of a different kind. For example, an update of every 500 ms could miss all speed updates because there is a wheel pressure update that resets the elapsed time. Producer services should, however, still implement a filtering scheme that could use proprietary attributes to filter on different kind of objects.

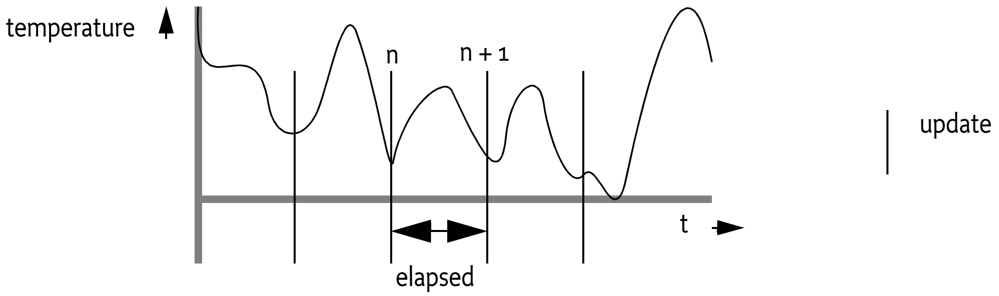
108.7.1 Filtering by Time

The simplest filter mechanism is based on time. The `wirevalue.elapsed` attribute contains the amount of milliseconds that have passed since the last update to the associated `Consumer` service. The following example filter expression illustrates how the updates can be limited to approximately 40 times per minute (once every 1500 ms).

```
(wirevalue.elapsed>=1500)
```

Figure 108.6 depicts this example graphically.

Figure 108.6 Elapsed Time Change



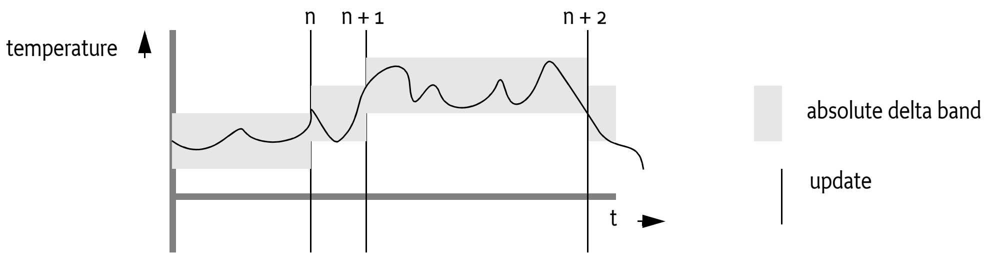
108.7.2 Filtering by Change

A Consumer service is often not interested in an update if the data value has not changed. The following filter expression shows how a Consumer service can limit the updates from a temperature sensor to be sent only when the temperature has changed at least 1 °K.

```
(wirevalue.delta.absolute>=1)
```

Figure 108.7 depicts a band that is created by the absolute delta between the previous data value and the current data value. The Consumer is only notified with the `updated(Wire,Object)` method when a data value is outside of this band.

Figure 108.7 Absolute Delta

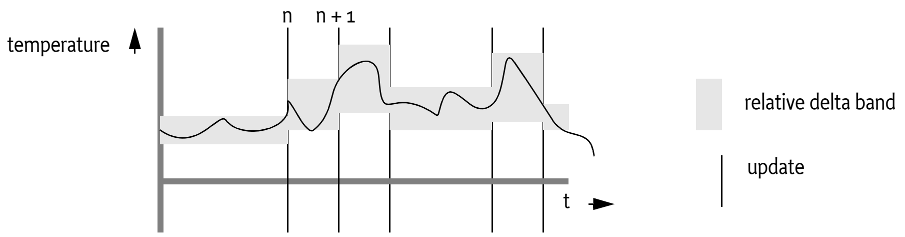


The delta may also be relative. For example, if a car is moving slowly, then updates for the speed of the car are interesting even for small variations. When a car is moving at a high rate of speed, updates are only interesting for larger variations in speed. The following example shows how the updates can be limited to data value changes of at least 10%.

```
(wirevalue.delta.relative>=0.1)
```

Figure 108.8 on page 198 depicts a relative band. Notice that the size of the band is directly proportional to the size of the sample value.

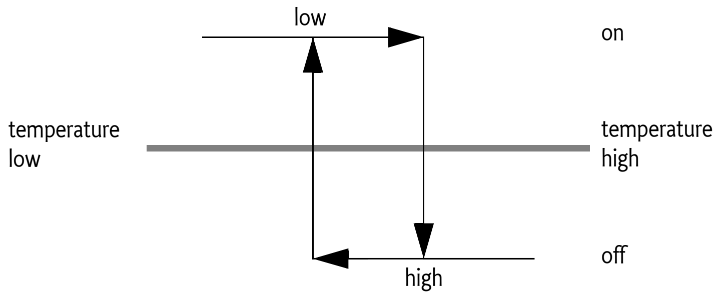
Figure 108.8 *Relative Delta (not to scale)*



108.7.3 Hysteresis

A thermostat is a control device that usually has a hysteresis, which means that a heater should be switched on below a certain specified low temperature and should be switched off at a specified high temperature, where *high* > *low*. This is graphically depicted in Figure 108.9 on page 198. The specified acceptable temperatures reduce the amount of start/stops of the heater.

Figure 108.9 *Hysteresis*



A Consumer service that controls the heater is only interested in events at the top and bottom of the hysteresis. If the specified high value is 250 °K and the specified low value is 249 °K, the following filter illustrates this concept:

```
(|(&(wirevalue.previous<=250) (wirevalue.current>250))
    (&(wirevalue.previous>=249) (wirevalue.current<249))
)
```

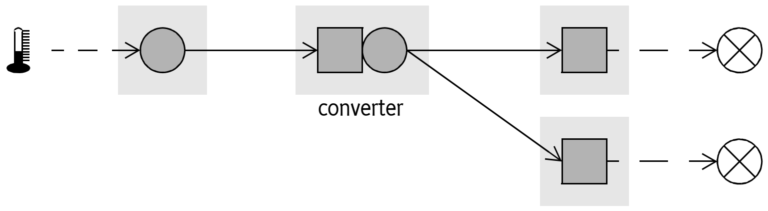
108.8 Flavors

Both Consumer and Producer services should register with a property describing the classes of the data types they can consume or produce respectively. The classes are the *flavors* that the service supports. The purpose of flavors is to allow an administrative user interface bundle to connect Consumer and Producer services. Bundles should only create a connection when there is at least one class shared between the flavors from a Consumer service and a Producer service. Producer services are responsible for selecting the preferred object type from the list of the object types preferred by the Consumer service. If the Producer service cannot convert its data to any of the flavors listed by the Consumer service, null should be used instead.

108.9 Converters

A converter is a bundle that registers a Consumer and a Producer service that are related and performs data conversions. Data values delivered to the Consumer service are processed and transferred via the related Producer service. The Producer service sends the converted data to other Consumer services. This is shown in Figure 108.10.

Figure 108.10 *Converter (for legend see Figure 108.2)*



108.10 Wire Admin Service Implementation

The Wire Admin service is the administrative service that is used to control the wiring topology in the OSGi Framework. It contains methods to create or update wires, delete wires, and list existing wires. It is intended to be used by user interfaces or management programs that control the wiring topology of the OSGi Framework.

The createWire(String,String,Dictionary) method is used to associate a Producer service with a Consumer service. The method always creates and returns a new object. It is therefore possible to create multiple, distinct wires between a Producer and a Consumer service. The properties can be used to create multiple associations between Producer and Consumer services in that act in different ways.

The properties of a Wire object can be updated with the update(Object) method. This method must update the properties in the Wire object and must notify the associated Consumer and Producer services if they are registered. Wire objects that are no longer needed can be removed with the deleteWire(Wire) method. All these methods are in the WireAdmin class and not in the Wire class for security reasons. See *Security* on page 202.

The getWires(String) method returns an array of Wire objects (or null). All objects are returned when the filter argument is null. Specifying a filter argument limits the returned objects. The filter uses the same syntax as the Framework Filter specification. This filter is applied to the properties of the Wire object and only Wire objects that match this filter are returned.

The following example shows how the getWires method can be used to print the PIDs of Producer services that are wired to a specific Consumer service.

```
String f = "(wireadmin.consumer.pid=com.acme.x)";
Wire [] wires = getWireAdmin().getWires( f );
for ( int i=0; wires != null && i < wires.length;i++ )
    System.out.println(
        wires[i].getProperties().get(
            "wireadmin.producer.pid")
    );
```

108.11 Wire Admin Listener Service Events

The Wire Admin service has an extensive list of events that it can deliver. The events allow other bundles to track changes in the topology as they happen. For example, a graphic user interface program can use the events to show when Wire objects become connected, when these objects are deleted, and when data flows over a Wire object.

A bundle that is interested in such events must register a WireAdminListener service object with a special Integer property WIREADMIN_EVENTS ("wireadmin.events"). This Integer object contains a bitmap of all the events in which this Wire Admin Listener service is interested (events have associated constants that can be ORed together). A Wire Admin service must not deliver events to the Wire Admin Listener service when that event type is not in the bitmap. If no such property is registered, no events are delivered to the Wire Admin Listener service.

The WireAdminListener interface has only one method: wireAdminEvent(WireAdminEvent). The argument is a WireAdminEvent object that contains the event type and associated data.

A WireAdminEvent object can be sent asynchronously but must be ordered for each Wire Admin Listener service. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 8*. Wire Admin Listener services must not assume that the state reflected by the event is still true when they receive the event.

The following types are defined for a WireEvent object:

Table 108.4 *Events*

Event type	Description
WIRE_CREATED	A new Wire object has been created.
WIRE_CONNECTED	Both the Producer service and the Consumer service are registered but may not have executed their respective connectedProducers/connectedConsumers methods.
WIRE_UPDATED	The Wire object's properties have been updated.
WIRE_TRACE	The Consumer has seen a new value, either after the Producer service has called the Wire. update(Object) method and the value was not filtered, or the Producer service has returned from the polled(Wire) method.
WIRE_DISCONNECTED	The Producer service or Consumer service have become unregistered and the Wire object is no longer connected.
WIRE_DELETED	The Wire object is deleted from the repository and is no longer available from the getWires method.
CONSUMER_EXCEPTION	The Consumer service generated an exception and the exception is included in the event.
PRODUCER_EXCEPTION	The Producer service generated an exception in a callback and the exception is included in the event.

108.11.1 Event Admin Service Events

Wire admin events must be sent asynchronously to the Event Admin service by the Wire Admin implementation, if present. The topic of a Wire Admin Event is one of the following:

```
org/osgi/service/wireadmin/WireAdminEvent/<eventtype>
```

The following event types are supported:

```
WIRE_CREATED
WIRE_CONNECTED
WIRE_UPDATED
WIRE_TRACE
WIRE_DISCONNECTED
WIRE_DELETED
PRODUCER_EXCEPTION
CONSUMER_EXCEPTION
```

The properties of a wire admin event are the following.

- event - (WireAdminEvent) The WireAdminEvent object broadcast by the Wire Admin service.

If the getWire method returns a non null value:

- wire - (Wire) The Wire object returned by the getWire method.
- wire.flavors - (String[]) The names of the classes returned by the Wire getFlavors method.
- wire.scope - (String[]) The scope of the Wire object, as returned by its getScope method.
- wire.connected - (Boolean) The result of the Wire isConnected method.
- wire.valid - (Boolean) The result of the Wire isValid method.

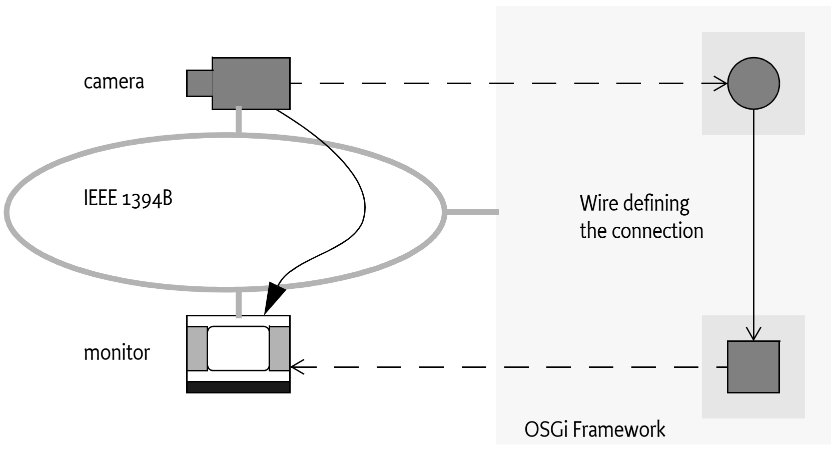
If the getThrowable method does not return null:

- exception - (Throwable) The Exception returned by the getThrowable method.
- exception.class - (String) The fully-qualified class name of the related Exception.
- exception.message - (String) The message of the related Exception
- service - (ServiceReference) The Service Reference of the Wire Admin service.
- service.id - (Long) The service id of the WireAdmin service.
- service.objectClass - (String[]) The Wire Admin service's object class (which must include org.osgi.service.wireadmin.WireAdmin)
- service.pid - (String) The Wire Admin service's PID.

108.12 Connecting External Entities

The Wire Admin service can be used to control the topology of consumers and producers that are services, as well as external entities. For example, a video camera controlled over an IEEE 1394B bus can be registered as a Producer service in the Framework's service registry and a TV, also connected to this bus, can be registered as a Consumer service. It would be very inefficient to stream the video data through the OSGi environment. Therefore, the Wire Admin service can be used to supply the external addressing information to the camera and the monitor to make a direct connection *outside* the OSGi environment. The Wire Admin service provides a uniform mechanism to connect both external entities and internal entities.

Figure 108.11 Connecting External Entities



A Consumer service and a Producer service associated with a Wire object receive enough information to establish a direct link because the PIDs of both services are in the Wire object's properties. This situation, however, does not guarantee *compatibility* between Producer and the Consumer service. It is therefore recommended that flavors are used to ensure this compatibility. Producer services that participate in an external addressing scheme, like IEEE 1394B, should have a flavor that reflects this address. In this case, there should then for example be a IEEE 1394B address class. Consumer services that participate in this external addressing scheme should only accept data of this flavor.

The OSGi *Device Access Specification* on page 39, defines the concept of a device category. This is a description of what classes and properties are used in a specific device category: for example, a UPnP device category that defines the interface that must be used to register for a UPnP device, among other things.

Device category descriptions should include a section that addresses the external wiring issue. This section should include what objects are send over the wire to exchange addressing information.

108.13 Related Standards

108.13.1 Java Beans

The Wire Admin service leverages the component architecture that the Framework service registry offers. Java Beans attempt to achieve similar goals. Java Beans are classes that follow a number of recommendations that allow them to be configured at run time. The techniques that are used by Java Beans during configuration are serialization and the construction of adapter classes.

Creating adapter classes in a resource constrained OSGi Framework was considered too heavy weight. Also, the dynamic nature of the OSGi environment, where services are registered and unregistered continuously, creates a mismatch between the intended target area of Java Beans and the OSGi Framework.

Also, Java Beans can freely communicate once they have a reference to each other. This freedom makes it impossible to control the communication between Java Beans.

This Wire Admin service specification was developed because it is lightweight and leverages the unique characteristics of the OSGi Framework. The concept of a `Wire` object that acts as an intermediate between the Producer and Consumer service allows the implementation of a security policy because both parties cannot communicate directly.

108.14 Security

108.14.1 Separation of Consumer and Producer Services

The Consumer and Producer service never directly communicate with each other. All communication takes place through a `Wire` object. This allows a Wire Admin service implementation to control the security aspects of creating a connection, and implies that the Wire Admin service must be a trusted service in a secure environment. Only one bundle should have the `ServicePermission[WireAdmin, REGISTER]`.

`ServicePermission[Producer|Consumer, REGISTER]` should not be restricted. `ServicePermission[Producer|Consumer,GET]` must be limited to trusted bundles (the Wire Admin service implementation) because a bundle with this permission can call such services and access information that it should not be able to access.

108.14.2 Using Wire Admin Service

This specification assumes that only a few applications require access to the Wire Admin service. The `WireAdmin` interface contains all the security sensitive methods that create, update, and remove `Wire` objects. (This is the reason that the update and delete methods are on the `WireAdmin` interface and not on the `Wire` interface). `ServicePermission[WireAdmin,GET]` should therefore only be given to trusted bundles that can manage the topology.

108.14.3 Wire Permission

Composite Producer and Consumer services can be restricted in their use of scope names. This restriction is managed with the WirePermission class. A WirePermission consists of a scope name and the action CONSUME or PRODUCE. The name used with the WirePermission may contain wild-cards as specified in the java.security.BasicPermission class.

108.15 org.osgi.service.wireadmin

Wire Admin Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.wireadmin; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.wireadmin; version="[1.0,1.1)"
```

108.15.1 Summary

- BasicEnvelope - BasicEnvelope is an implementation of the Envelope interface
- Consumer - Data Consumer, a service that can receive updated values from Producer services.
- Envelope - Identifies a contained value.
- Producer - Data Producer, a service that can generate values to be used by Consumer services.
- Wire - A connection between a Producer service and a Consumer service.
- WireAdmin - Wire Administration service.
- WireAdminEvent - A Wire Admin Event.
- WireAdminListener - Listener for Wire Admin Events.
- WireConstants - Defines standard names for Wire properties, wire filter attributes, Consumer and Producer service properties.
- WirePermission - Permission for the scope of a Wire object.

108.15.2 public class BasicEnvelope implements Envelope

BasicEnvelope is an implementation of the Envelope interface

Concurrency Immutable

108.15.2.1 public BasicEnvelope(Object value, Object identification, String scope)

value Content of this envelope, may be null.

identification Identifying object for this Envelope object, must not be null

scope Scope name for this object, must not be null

□ Constructor.

See Also Envelope

108.15.2.2 public Object getIdentification()

□ Return the identification of this Envelope object. An identification may be of any Java type. The type must be mutually agreed between the Consumer and Producer services.

Returns an object which identifies the status item in the address space of the composite producer, must not be null.

See Also org.osgi.service.wireadmin.Envelope.getIdentification()

108.15.2.3 public String getScope()

□ Return the scope name of this Envelope object. Scope names are used to restrict the communication between the Producer and Consumer services. Only Envelopes objects with a scope name that is permitted for the Producer and the Consumer services must be passed through a Wire object.

Returns the security scope for the status item, must not be null.

See Also org.osgi.service.wireadmin.Envelope.getScope()

108.15.2.4 public Object getValue()

□ Return the value associated with this Envelope object.

Returns the value of the status item, or null when no item is associated with this object.

See Also org.osgi.service.wireadmin.Envelope.getValue()

108.15.3 public interface Consumer

Data Consumer, a service that can receive updated values from Producer services.

Service objects registered under the Consumer interface are expected to consume values from a Producer service via a Wire object. A Consumer service may poll the Producer service by calling the Wire.poll() method. The Consumer service will also receive an updated value when called at it's updated(Wire, Object) method. The Producer service should have coerced the value to be an instance of one of the types specified by the Wire.getFlavors() method, or one of their subclasses.

Consumer service objects must register with a service.pid and a WireConstants.WIREADMIN_CONSUMER_FLAVORS property. It is recommended that Consumer service objects also register with a service.description property.

If an Exception is thrown by any of the Consumer methods, a WireAdminEvent of type WireAdminEvent.CONSUMER_EXCEPTION is broadcast by the Wire Admin service.

Security Considerations - Data consuming bundles will require ServicePermission[Consumer,REGISTER]. In general, only the Wire Admin service bundle should have this permission. Thus only the Wire Admin service may directly call a Consumer service. Care must be taken in the sharing of Wire objects with other bundles.

Consumer services must be registered with their scope when they can receive different types of objects from the Producer service. The Consumer service should have WirePermission for each of these scope names.

108.15.3.1 public void producersConnected(Wire[] wires)

wires An array of the current and complete list of Wire objects to which this Consumer service is connected. May be null if the Consumer service is not currently connected to any Wire objects.

□ Update the list of Wire objects to which this Consumer service is connected.

This method is called when the Consumer service is first registered and subsequently whenever a Wire associated with this Consumer service becomes connected, is modified or becomes disconnected.

The Wire Admin service must call this method asynchronously. This implies that implementors of Consumer can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

108.15.3.2 public void updated(Wire wire, Object value)

wire The Wire object which is delivering the updated value.

value The updated value. The value should be an instance of one of the types specified by the Wire.getFlavors() method.

□ Update the value. This Consumer service is called by the Wire object with an updated value from the Producer service.

Note: This method may be called by a Wire object prior to this object being notified that it is connected to that Wire object (via the producersConnected(Wire[]) method).

When the Consumer service can receive Envelope objects, it must have registered all scope names together with the service object, and each of those names must be permitted by the bundle's WirePermission. If an Envelope object is delivered with the updated method, then the Consumer service should assume that the security check has been performed.

108.15.4 public interface Envelope

Identifies a contained value. An Envelope object combines a status value, an identification object and a scope name. The Envelope object allows the use of standard Java types when a Producer service can produce more than one kind of object. The Envelope object allows the Consumer service to recognize the kind of object that is received. For example, a door lock could be represented by a Boolean object. If the Producer service would send such a Boolean object, then the Consumer service would not know what door the Boolean object represented. The Envelope object contains an identification object so the Consumer service can discriminate between different kinds of values. The identification object may be a simple String object, but it can also be a domain specific object that is mutually agreed by the Producer and the Consumer service. This object can then contain relevant information that makes the identification easier.

The scope name of the envelope is used for security. The Wire object must verify that any Envelope object send through the update method or coming from the poll method has a scope name that matches the permissions of both the Producer service and the Consumer service involved. The wireadmin package also contains a class BasicEnvelope that implements the methods of this interface.

See Also WirePermission, BasicEnvelope

108.15.4.1 public Object getIdentification()

□ Return the identification of this Envelope object. An identification may be of any Java type. The type must be mutually agreed between the Consumer and Producer services.

Returns an object which identifies the status item in the address space of the composite producer, must not be null.

108.15.4.2 public String getScope()

□ Return the scope name of this Envelope object. Scope names are used to restrict the communication between the Producer and Consumer services. Only Envelopes objects with a scope name that is permitted for the Producer and the Consumer services must be passed through a Wire object.

Returns the security scope for the status item, must not be null.

108.15.4.3 public Object getValue()

□ Return the value associated with this Envelope object.

Returns the value of the status item, or null when no item is associated with this object.

108.15.5 public interface Producer

Data Producer, a service that can generate values to be used by Consumer services.

Service objects registered under the Producer interface are expected to produce values (internally generated or from external sensors). The value can be of different types. When delivering a value to a Wire object, the Producer service should coerce the value to be an instance of one of the types specified by Wire.getFlavors(). The classes are specified in order of preference.

When the data represented by the Producer object changes, this object should send the updated value by calling the update method on each of Wire objects passed in the most recent call to this object's consumersConnected(Wire[]) method. These Wire objects will pass the value on to the associated Consumer service object.

The Producer service may use the information in the Wire object's properties to schedule the delivery of values to the Wire object.

Producer service objects must register with a service.pid and a WireConstants.WIREADMIN_PRODUCER_FLAVORS property. It is recommended that a Producer service object also registers with a service.description property. Producer service objects must register with a WireConstants.WIREADMIN_PRODUCER_FILTERS property if the Producer service will be performing filtering instead of the Wire object.

If an exception is thrown by a Producer object method, a WireAdminEvent of type WireAdminEvent.PRODUCER_EXCEPTION is broadcast by the Wire Admin service.

Security Considerations. Data producing bundles will require ServicePermission[Producer,REGISTER] to register a Producer service. In general, only the Wire Admin service should have ServicePermission[Producer,GET]. Thus only the Wire Admin service may directly call a Producer service. Care must be taken in the sharing of Wire objects with other bundles.

Producer services must be registered with scope names when they can send different types of objects (composite) to the Consumer service. The Producer service should have WirePermission for each of these scope names.

108.15.5.1 public void consumersConnected(Wire[] wires)

wires An array of the current and complete list of Wire objects to which this Producer service is connected. May be null if the Producer is not currently connected to any Wire objects.

□ Update the list of Wire objects to which this Producer object is connected.

This method is called when the Producer service is first registered and subsequently whenever a Wire associated with this Producer becomes connected, is modified or becomes disconnected.

The Wire Admin service must call this method asynchronously. This implies that implementors of a Producer service can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

108.15.5.2 public Object polled(Wire wire)

wire The Wire object which is polling this service.

□ Return the current value of this Producer object.

This method is called by a Wire object in response to the Consumer service calling the Wire object's poll method. The Producer should coerce the value to be an instance of one of the types specified by Wire.getFlavors(). The types are specified in order of preference. The returned value should be as new or newer than the last value furnished by this object.

Note: This method may be called by a Wire object prior to this object being notified that it is connected to that Wire object (via the consumersConnected(Wire[]) method).

If the Producer service returns an Envelope object that has an impermissible scope name, then the Wire object must ignore (or remove) the transfer.

If the Wire object has a scope set, the return value must be an array of Envelope objects (Envelope[]). The Wire object must have removed any Envelope objects that have a scope name that is not in the Wire object's scope.

Returns The current value of the Producer service or null if the value cannot be coerced into a compatible type. Or an array of Envelope objects.

108.15.6 public interface Wire

A connection between a Producer service and a Consumer service.

A Wire object connects a Producer service to a Consumer service. Both the Producer and Consumer services are identified by their unique service.pid values. The Producer and Consumer services may communicate with each other via Wire objects that connect them. The Producer service may send updated values to the Consumer service by calling the update(Object) method. The Consumer service may request an updated value from the Producer service by calling the poll() method.

A Producer service and a Consumer service may be connected through multiple Wire objects.

Security Considerations. Wire objects are available to Producer and Consumer services connected to a given Wire object and to bundles which can access the WireAdmin service. A bundle must have ServicePermission[WireAdmin,GET] to get the WireAdmin service to access all Wire objects. A bundle registering a Producer service or a Consumer service must have the appropriate ServicePermission[Consumer|Producer,REGISTER] to register the service and will be passed Wire objects when the service object's consumersConnected or producersConnected method is called.

Scope. Each Wire object can have a scope set with the setScope method. This method should be called by a Consumer service when it assumes a Producer service that is composite (supports multiple information items). The names in the scope must be verified by the Wire object before it is used in communication. The semantics of the names depend on the Producer service and must not be interpreted by the Wire Admin service.

No Implement Consumers of this API must not implement this interface

108.15.6.1 public Class<?>[] getFlavors()

□ Return the list of data types understood by the Consumer service connected to this Wire object. Note that subclasses of the classes in this list are acceptable data types as well.

The list is the value of the WireConstants.WIREADMIN_CONSUMER_FLAVORS service property of the Consumer service object connected to this object. If no such property was registered or the type of the property value is not Class[], this method must return null.

Returns An array containing the list of classes understood by the Consumer service or null if the Wire is not connected, or the consumer did not register a WireConstants.WIREADMIN_CONSUMER_FLAVORS property or the value of the property is not of type Class[].

108.15.6.2 public Object getLastValue()

□ Return the last value sent through this Wire object.

The returned value is the most recent, valid value passed to the update(Object) method or returned by the poll() method of this object. If filtering is performed by this Wire object, this methods returns the last value provided by the Producer service. This value may be an Envelope[] when the Producer service uses scoping. If the return value is an Envelope object (or array), it must be verified that the Consumer service has the proper WirePermission to see it.

Returns The last value passed though this Wire object or null if no valid values have been passed or the Consumer service has no permission.

108.15.6.3 public Dictionary<String, Object> getProperties()

□ Return the wire properties for this Wire object.

Returns The properties for this Wire object. The returned Dictionary must be read only.

108.15.6.4 public String[] getScope()

□ Return the calculated scope of this Wire object. The purpose of the Wire object's scope is to allow a Producer and/or Consumer service to produce/consume different types over a single Wire object (this was deemed necessary for efficiency reasons). Both the Consumer service and the Producer service must set an array of scope names (their scope) with the service registration property WIREADMIN_PRODUCER_SCOPE, or WIREADMIN_CONSUMER_SCOPE when they can produce multiple types. If a Producer service can produce different types, it should set this property to the array of scope names it can produce, the Consumer service must set the array of scope names it can consume. The scope of a Wire object is defined as the intersection of permitted scope names of the Producer service and Consumer service.

If neither the Consumer, or the Producer service registers scope names with its service registration, then the Wire object's scope must be null.

The Wire object's scope must not change when a Producer or Consumer services modifies its scope.

A scope name is permitted for a Producer service when the registering bundle has WirePermission[name,PRODUCE], and for a Consumer service when the registering bundle has WirePermission[name,CONSUME].

If either Consumer service or Producer service has not set a WIREADMIN_*_SCOPE property, then the returned value must be null.

If the scope is set, the Wire object must enforce the scope names when Envelope objects are used as a parameter to update or returned from the poll method. The Wire object must then remove all Envelope objects with a scope name that is not permitted.

Returns A list of permitted scope names or null if the Produce or Consumer service has set no scope names.

108.15.6.5 public boolean hasScope(String name)

name The scope name

□ Return true if the given name is in this Wire object's scope.

Returns true if the name is listed in the permitted scope names

108.15.6.6 public boolean isConnected()

□ Return the connection state of this Wire object.

A Wire is connected after the Wire Admin service receives notification that the Producer service and the Consumer service for this Wire object are both registered. This method will return true prior to notifying the Producer and Consumer services via calls to their respective consumersConnected and producersConnected methods.

A WireAdminEvent of type WireAdminEvent.WIRE_CONNECTED must be broadcast by the Wire Admin service when the Wire becomes connected.

A Wire object is disconnected when either the Consumer or Producer service is unregistered or the Wire object is deleted.

A WireAdminEvent of type WireAdminEvent.WIRE_DISCONNECTED must be broadcast by the Wire Admin service when the Wire becomes disconnected.

Returns true if both the Producer and Consumer for this Wire object are connected to the Wire object; false otherwise.

108.15.6.7 public boolean isValid()

□ Return the state of this Wire object.

A connected Wire must always be disconnected before becoming invalid.

Returns false if this Wire object is invalid because it has been deleted via WireAdmin.deleteWire(Wire); true otherwise.

108.15.6.8 public Object poll()

□ Poll for an updated value.

This methods is normally called by the Consumer service to request an updated value from the Producer service connected to this Wire object. This Wire object will call the Producer.polled(Wire) method to obtain an updated value. If this Wire object is not connected, then the Producer service must not be called.

If this Wire object has a scope, then this method must return an array of Envelope objects. The objects returned must match the scope of this object. The Wire object must remove all Envelope objects with a scope name that is not in the Wire object's scope. Thus, the list of objects returned must only contain Envelope objects with a permitted scope name. If the array becomes empty, null must be returned.

A WireAdminEvent of type WireAdminEvent.WIRE_TRACE must be broadcast by the Wire Admin service after the Producer service has been successfully called.

Returns A value whose type should be one of the types returned by getFlavors(),Envelope[], or null if the Wire object is not connected, the Producer service threw an exception, or the Producer service returned a value which is not an instance of one of the types returned by getFlavors().

108.15.6.9 public void update(Object value)

value The updated value. The value should be an instance of one of the types returned by getFlavors().

□ Update the value.

This methods is called by the Producer service to notify the Consumer service connected to this Wire object of an updated value.

If the properties of this Wire object contain a WireConstants.WIREADMIN_FILTER property, then filtering is performed. If the Producer service connected to this Wire object was registered with the service property WireConstants.WIREADMIN_PRODUCER_FILTERS, the Producer service will perform the filtering according to the rules specified for the filter. Otherwise, this Wire object will perform the filtering of the value.

If no filtering is done, or the filter indicates the updated value should be delivered to the Consumer service, then this Wire object must call the Consumer.updated(Wire, Object) method with the updated value. If this Wire object is not connected, then the Consumer service must not be called and the value is ignored.

If the value is an Envelope object, and the scope name is not permitted, then the Wire object must ignore this call and not transfer the object to the Consumer service.

A WireAdminEvent of type WireAdminEvent.WIRE_TRACE must be broadcast by the Wire Admin service after the Consumer service has been successfully called.

See Also WireConstants.WIREADMIN_FILTER

108.15.7 public interface WireAdmin

Wire Administration service.

This service can be used to create Wire objects connecting a Producer service and a Consumer service. Wire objects also have wire properties that may be specified when a Wire object is created. The

Producer and Consumer services may use the Wire object's properties to manage or control their interaction. The use of Wire object's properties by a Producer or Consumer services is optional.

Security Considerations. A bundle must have ServicePermission[WireAdmin,GET] to get the Wire Admin service to create, modify, find, and delete Wire objects.

No Implement Consumers of this API must not implement this interface

108.15.7.1 public Wire createWire(String producerPID, String consumerPID, Dictionary<String, ?> properties)

producerPID The service.pid of the Producer service to be connected to the Wire object.

consumerPID The service.pid of the Consumer service to be connected to the Wire object.

properties The Wire object's properties. This argument may be null if the caller does not wish to define any Wire object's properties.

□ Create a new Wire object that connects a Producer service to a Consumer service. The Producer service and Consumer service do not have to be registered when the Wire object is created.

The Wire configuration data must be persistently stored. All Wire connections are reestablished when the WireAdmin service is registered. A Wire can be permanently removed by using the deleteWire(Wire) method.

The Wire object's properties must have case insensitive String objects as keys (like the Framework). However, the case of the key must be preserved.

The WireAdmin service must automatically add the following Wire properties:

- WireConstants.WIREADMIN_PID set to the value of the Wire object's persistent identity (PID). This value is generated by the Wire Admin service when a Wire object is created.
- WireConstants.WIREADMIN_PRODUCER_PID set to the value of Producer service's PID.
- WireConstants.WIREADMIN_CONSUMER_PID set to the value of Consumer service's PID.

If the properties argument already contains any of these keys, then the supplied values are replaced with the values assigned by the Wire Admin service.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_CREATED after the new Wire object becomes available from getWires(String).

Returns The Wire object for this connection.

Throws IllegalArgumentException– If properties contains invalid wire types or case variants of the same key name.

108.15.7.2 public void deleteWire(Wire wire)

wire The Wire object which is to be deleted.

□ Delete a Wire object.

The Wire object representing a connection between a Producer service and a Consumer service must be removed. The persistently stored configuration data for the Wire object must destroyed. The Wire object's method Wire.isValid() will return false after it is deleted.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_DELETED after the Wire object becomes invalid.

108.15.7.3 public Wire[] getWires(String filter) throws InvalidSyntaxException

filter Filter string to select Wire objects or null to select all Wire objects.

□ Return the Wire objects that match the given filter.

The list of available Wire objects is matched against the specified filter.Wire objects which match the filter must be returned. These Wire objects are not necessarily connected. The Wire Admin service should not return invalid Wire objects, but it is possible that a Wire object is deleted after it was placed in the list.

The filter matches against the Wire object's properties including WireConstants.WIREADMIN_PRODUCER_PID, WireConstants.WIREADMIN_CONSUMER_PID and WireConstants.WIREADMIN_PID.

Returns An array of Wire objects which match the filter or null if no Wire objects match the filter.

Throws InvalidSyntaxException– If the specified filter has an invalid syntax.

See Also org.osgi.framework.Filter

108.15.7.4 public void updateWire(Wire wire, Dictionary<String, ?> properties)

wire The Wire object which is to be updated.

properties The new Wire object's properties or null if no properties are required.

□ Update the properties of a Wire object. The persistently stored configuration data for the Wire object is updated with the new properties and then the Consumer and Producer services will be called at the respective Consumer.producersConnected(Wire[]) and Producer.consumersConnected(Wire[]) methods.

The Wire Admin service must broadcast a WireAdminEvent of type WireAdminEvent.WIRE_UPDATED after the updated properties are available from the Wire object.

Throws IllegalArgumentException– If properties contains invalid wire types or case variants of the same key name.

108.15.8 public class WireAdminEvent

A Wire Admin Event.

WireAdminEvent objects are delivered to all registered WireAdminListener service objects which specify an interest in the WireAdminEvent type. Events must be delivered in chronological order with respect to each listener. For example, a WireAdminEvent of type WIRE_CONNECTED must be delivered before a WireAdminEvent of type WIRE_DISCONNECTED for a particular Wire object.

A type code is used to identify the type of event. The following event types are defined:

- WIRE_CREATED
- WIRE_CONNECTED
- WIRE_UPDATED
- WIRE_TRACE
- WIRE_DISCONNECTED
- WIRE_DELETED
- PRODUCER_EXCEPTION
- CONSUMER_EXCEPTION

Additional event types may be defined in the future.

Event type values must be unique and disjoint bit values. Event types must be defined as a bit in a 32 bit integer and can thus be bitwise ORed together.

Security Considerations. WireAdminEvent objects contain Wire objects. Care must be taken in the sharing of Wire objects with other bundles.

See Also WireAdminListener

Concurrency Immutable

108.15.8.1 public static final int CONSUMER_EXCEPTION = 2

A Consumer service method has thrown an exception.

This WireAdminEvent type indicates that a Consumer service method has thrown an exception. The WireAdminEvent.getThrowable() method will return the exception that the Consumer service method raised.

The value of CONSUMER_EXCEPTION is 0x00000002.

108.15.8.2 public static final int PRODUCER_EXCEPTION = 1

A Producer service method has thrown an exception.

This WireAdminEvent type indicates that a Producer service method has thrown an exception. The WireAdminEvent.getThrowable() method will return the exception that the Producer service method raised.

The value of PRODUCER_EXCEPTION is 0x00000001.

108.15.8.3 public static final int WIRE_CONNECTED = 32

The WireAdminEvent type that indicates that an existing Wire object has become connected. The Consumer object and the Producer object that are associated with the Wire object have both been registered and the Wire object is connected. See Wire.isConnected() for a description of the connected state. This event may come before the producersConnected and consumersConnected method have returned or called to allow synchronous delivery of the events. Both methods can cause other WireAdminEvent s to take place and requiring this event to be send before these methods are returned would mandate asynchronous delivery.

The value of WIRE_CONNECTED is 0x00000020.

108.15.8.4 public static final int WIRE_CREATED = 4

A Wire has been created.

This WireAdminEvent type that indicates that a new Wire object has been created. An event is broadcast when WireAdmin.createWire(String, String, java.util.Dictionary) is called. The WireAdminEvent.getWire() method will return the Wire object that has just been created.

The value of WIRE_CREATED is 0x00000004.

108.15.8.5 public static final int WIRE_DELETED = 16

A Wire has been deleted.

This WireAdminEvent type that indicates that an existing wire has been deleted. An event is broadcast when WireAdmin.deleteWire(Wire) is called with a valid wire. WireAdminEvent.getWire() will return the Wire object that has just been deleted.

The value of WIRE_DELETED is 0x00000010.

108.15.8.6 public static final int WIRE_DISCONNECTED = 64

The WireAdminEvent type that indicates that an existing Wire object has become disconnected. The Consumer object or/and Producer object is/are unregistered breaking the connection between the two. See Wire.isConnected for a description of the connected state.

The value of WIRE_DISCONNECTED is 0x00000040.

108.15.8.7 public static final int WIRE_TRACE = 128

The WireAdminEvent type that indicates that a new value is transferred over the Wire object. This event is sent after the Consumer service has been notified by calling the Consumer.updated(Wire, Object) method or the Consumer service requested a new value with the Wire.poll() method. This is an advisory event meaning that when this event is received, another update may already have occurred and this the Wire.getLastValue() method returns a newer value then the value that was communicated for this event.

The value of WIRE_TRACE is 0x00000080.

108.15.8.8 public static final int WIRE_UPDATED = 8

A Wire has been updated.

This WireAdminEvent type that indicates that an existing Wire object has been updated with new properties. An event is broadcast when WireAdmin.updateWire(Wire, java.util.Dictionary) is called with a valid wire. The WireAdminEvent.getWire() method will return the Wire object that has just been updated.

The value of WIRE_UPDATED is 0x00000008.

108.15.8.9 public WireAdminEvent(ServiceReference<WireAdmin> reference, int type, Wire wire, Throwable exception)

reference The ServiceReference object of the Wire Admin service that created this event.

type The event type. See getType().

wire The Wire object associated with this event.

exception An exception associated with this event. This may be null if no exception is associated with this event.

□ Constructs a WireAdminEvent object from the given ServiceReference object, event type, Wire object and exception.

108.15.8.10 public ServiceReference<WireAdmin> getServiceReference()

□ Return the ServiceReference object of the Wire Admin service that created this event.

Returns The ServiceReference object for the Wire Admin service that created this event.

108.15.8.11 public Throwable getThrowable()

□ Returns the exception associated with the event, if any.

Returns An exception or null if no exception is associated with this event.

108.15.8.12 public int getType()

□ Return the type of this event.

The type values are:

- WIRE_CREATED
- WIRE_CONNECTED
- WIRE_UPDATED
- WIRE_TRACE
- WIRE_DISCONNECTED
- WIRE_DELETED
- PRODUCER_EXCEPTION
- CONSUMER_EXCEPTION

Returns The type of this event.

108.15.8.13 public Wire getWire()

□ Return the Wire object associated with this event.

Returns The Wire object associated with this event or null when no Wire object is associated with the event.

108.15.9 public interface WireAdminListener

Listener for Wire Admin Events.

WireAdminListener objects are registered with the Framework service registry and are notified with a WireAdminEvent object when an event is broadcast.

WireAdminListener objects can inspect the received WireAdminEvent object to determine its type, the Wire object with which it is associated, and the Wire Admin service that broadcasts the event.

WireAdminListener objects must be registered with a service property WireConstants.WIREADMIN_EVENTS whose value is a bitwise OR of all the event types the listener is interested in receiving.

For example:

```
Integer mask = Integer.valueOf(WIRE_TRACE | WIRE_CONNECTED | WIRE_DISCONNECTED);
Hashtable ht = new Hashtable();
ht.put(WIREADMIN_EVENTS, mask);
context.registerService(WireAdminListener.class.getName(), this, ht);
```

If a WireAdminListener object is registered without a service property WireConstants.WIREADMIN_EVENTS, then the WireAdminListener will receive no events.

Security Considerations. Bundles wishing to monitor WireAdminEvent objects will require ServicePermission[WireAdminListener,REGISTER] to register a WireAdminListener service. Since WireAdminEvent objects contain Wire objects, care must be taken in assigning permission to register a WireAdminListener service.

See Also WireAdminEvent

108.15.9.1 public void wireAdminEvent(WireAdminEvent event)

event The WireAdminEvent object.

□ Receives notification of a broadcast WireAdminEvent object. The event object will be of an event type specified in this WireAdminListener service's WireConstants.WIREADMIN_EVENTS service property.

108.15.10 public interface WireConstants

Defines standard names for Wire properties, wire filter attributes, Consumer and Producer service properties.

No Implement Consumers of this API must not implement this interface

108.15.10.1 public static final String WIREADMIN_CONSUMER_COMPOSITE = "wireadmin.consumer.composite"

A service registration property for a Consumer service that is composite. It contains the names of the composite Producer services it can cooperate with. Interoperability exists when any name in this array matches any name in the array set by the Producer service. The type of this property must be String[].

108.15.10.2 public static final String WIREADMIN_CONSUMER_FLAVORS = "wireadmin.consumer.flavors"

Service Registration property (named wireadmin.consumer.flavors) specifying the list of data types understood by this Consumer service.

The Consumer service object must be registered with this service property. The list must be in the order of preference with the first type being the most preferred. The value of the property must be of type Class[].

108.15.10.3 public static final String WIREADMIN_CONSUMER_PID = "wireadmin.consumer.pid"

Wire property key (named wireadmin.consumer.pid) specifying the service.pid of the associated Consumer service.

This wire property is automatically set by the Wire Admin service. The value of the property must be of type String.

108.15.10.4 public static final String WIREADMIN_CONSUMER_SCOPE = "wireadmin.consumer.scope"

Service registration property key (named wireadmin.consumer.scope) specifying a list of names that may be used to define the scope of this Wire object. A Consumer service should set this service property when it can produce more than one kind of value. This property is only used during registration, modifying the property must not have any effect of the Wire object's scope. Each name in the given list mist have WirePermission[name,CONSUME] or else is ignored. The type of this service registration property must be String[].

See Also Wire.getScope(), WIREADMIN_PRODUCER_SCOPE

108.15.10.5 public static final String WIREADMIN_EVENTS = "wireadmin.events"

Service Registration property (named wireadmin.events) specifying the WireAdminEvent type of interest to a Wire Admin Listener service. The value of the property is a bitwise OR of all the WireAdminEvent types the Wire Admin Listener service wishes to receive and must be of type Integer.

See Also WireAdminEvent

108.15.10.6 public static final String WIREADMIN_FILTER = "wireadmin.filter"

Wire property key (named wireadmin.filter) specifying a filter used to control the delivery rate of data between the Producer and the Consumer service.

This property should contain a filter as described in the Filter class. The filter can be used to specify when an updated value from the Producer service should be delivered to the Consumer service. In many cases the Consumer service does not need to receive the data with the same rate that the Producer service can generate data. This property can be used to control the delivery rate.

The filter can use a number of predefined attributes that can be used to control the delivery of new data values. If the filter produces a match upon the wire filter attributes, the Consumer service should be notified of the updated data value.

If the Producer service was registered with the WIREADMIN_PRODUCER_FILTERS service property indicating that the Producer service will perform the data filtering then the Wire object will not perform data filtering. Otherwise, the Wire object must perform basic filtering. Basic filtering includes supporting the following standard wire filter attributes:

- WIREVALUE_CURRENT - Current value
- WIREVALUE_PREVIOUS - Previous value
- WIREVALUE_DELTA_ABSOLUTE - Absolute delta
- WIREVALUE_DELTA_RELATIVE - Relative delta
- WIREVALUE_ELAPSED - Elapsed time

See Also org.osgi.framework.Filter

108.15.10.7 public static final String WIREADMIN_PID = "wireadmin.pid"

Wire property key (named wireadmin.pid) specifying the persistent identity (PID) of this Wire object.

Each Wire object has a PID to allow unique and persistent identification of a specific Wire object. The PID must be generated by the WireAdmin service when the Wire object is created.

This wire property is automatically set by the Wire Admin service. The value of the property must be of type String.

108.15.10.8 public static final String WIREADMIN_PRODUCER_COMPOSITE = "wireadmin.producer.composite"

A service registration property for a Producer service that is composite. It contains the names of the composite Consumer services it can interoperate with. Interoperability exists when any name in this array matches any name in the array set by the Consumer service. The type of this property must be String[].

108.15.10.9 public static final String WIREADMIN_PRODUCER_FILTERS = "wireadmin.producer.filters"

Service Registration property (named wireadmin.producer.filters). A Producer service registered with this property indicates to the Wire Admin service that the Producer service implements at least the filtering as described for the WIREADMIN_FILTER property. If the Producer service is not registered with this property, the Wire object must perform the basic filtering as described in WIREADMIN_FILTER.

The type of the property value is not relevant. Only its presence is relevant.

108.15.10.10 public static final String WIREADMIN_PRODUCER_FLAVORS = "wireadmin.producer.flavors"

Service Registration property (named wireadmin.producer.flavors) specifying the list of data types available from this Producer service.

The Producer service object should be registered with this service property.

The value of the property must be of type Class[].

108.15.10.11 public static final String WIREADMIN_PRODUCER_PID = "wireadmin.producer.pid"

Wire property key (named wireadmin.producer.pid) specifying the service.pid of the associated Producer service.

This wire property is automatically set by the WireAdmin service. The value of the property must be of type String.

108.15.10.12 public static final String WIREADMIN_PRODUCER_SCOPE = "wireadmin.producer.scope"

Service registration property key (named wireadmin.producer.scope) specifying a list of names that may be used to define the scope of this Wire object. A Producer service should set this service property when it can produce more than one kind of value. This property is only used during registration, modifying the property must not have any effect of the Wire object's scope. Each name in the given list mist have WirePermission[name,PRODUCE] or else is ignored. The type of this service registration property must be String[].

See Also Wire.getScope(), WIREADMIN_CONSUMER_SCOPE

108.15.10.13 public static final String[] WIREADMIN_SCOPE_ALL

Matches all scope names.

108.15.10.14 public static final String WIREVALUE_CURRENT = "wirevalue.current"

Wire object's filter attribute (named wirevalue.current) representing the current value.

108.15.10.15 public static final String WIREVALUE_DELTA_ABSOLUTE = "wirevalue.delta.absolute"

Wire object's filter attribute (named wirevalue.delta.absolute) representing the absolute delta. The absolute (always positive) difference between the last update and the current value (only when numeric). This attribute must not be used when the values are not numeric.

108.15.10.16 public static final String WIREVALUE_DELTA_RELATIVE = "wirevalue.delta.relative"

Wire object's filter attribute (named wirevalue.delta.relative) representing the relative delta. The relative difference is |previous-current |/| current| (only when numeric). This attribute must not be used when the values are not numeric.

108.15.10.17 public static final String WIREVALUE_ELAPSED = "wirevalue.elapsed"

Wire object's filter attribute (named wirevalue.elapsed) representing the elapsed time, in ms, between this filter evaluation and the last update of the Consumer service.

108.15.10.18 public static final String WIREVALUE_PREVIOUS = "wirevalue.previous"

Wire object's filter attribute (named wirevalue.previous) representing the previous value.

108.15.11 public final class WirePermission extends BasicPermission

Permission for the scope of a Wire object. When a Envelope object is used for communication with the poll or update method, and the scope is set, then the Wire object must verify that the Consumer service has WirePermission[name,CONSUME] and the Producer service has WirePermission[name,PRODUCE] for all names in the scope.

The names are compared with the normal rules for permission names. This means that they may end with a "*" to indicate wildcards. E.g. Door.* indicates all scope names starting with the string "Door". The last period is required due to the implementations of the BasicPermission class.

Concurrency Thread-safe

108.15.11.1 public static final String CONSUME = "consume"

The action string for the consume action.

108.15.11.2 public static final String PRODUCE = "produce"

The action string for the produce action.

108.15.11.3 public WirePermission(String name, String actions)

name Wire name.

actions produce, consume (canonical order).

□ Create a new WirePermission with the given name (may be wildcard) and actions.

108.15.11.4 public boolean equals(Object obj)

obj The object to test for equality.

□ Determines the equality of two WirePermission objects. Checks that specified object has the same name and actions as this WirePermission object.

Returns true if obj is a WirePermission, and has the same name and actions as this WirePermission object; false otherwise.

108.15.11.5 public String getActions()

□ Returns the canonical string representation of the actions. Always returns present actions in the following order: produce, consume.

Returns The canonical string representation of the actions.

108.15.11.6 public int hashCode()

□ Returns the hash code value for this object.

Returns Hash code value for this object.

108.15.11.7 public boolean implies(Permission p)

p The permission to check against.

□ Checks if this `WirePermission` object implies the specified permission.

More specifically, this method returns `true` if:

- *p* is an instanceof the `WirePermission` class,
- *p*'s actions are a proper subset of this object's actions, and
- *p*'s name is implied by this object's name. For example, `java.*` implies `java.home`.

Returns `true` if the specified permission is implied by this object; `false` otherwise.

108.15.11.8 public PermissionCollection newPermissionCollection()

□ Returns a new `PermissionCollection` object for storing `WirePermission` objects.

Returns A new `PermissionCollection` object suitable for storing `WirePermission` objects.

108.15.11.9 public String toString()

□ Returns a string describing this `WirePermission`. The convention is to specify the class name, the permission name, and the actions in the following format: '(org.osgi.service.wireadmin.WirePermission "name" "actions")'.

Returns information about this `Permission` object.

108.16 References

[1] *Design Patterns*
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley, ISBN 0-201-63361

111 Device Service Specification for UPnP™ Technology

Version 1.2

111.1 Introduction

The UPnP Device Architecture specification provides the protocols for a peer-to-peer network. It specifies how to join a network and how devices can be controlled using XML messages sent over HTTP. The OSGi specifications address how code can be download and managed in a remote system. Both standards are therefore fully complimentary. Using an OSGi Framework to work with UPnP enabled devices is therefore a very successful combination.

This specification specifies how OSGi bundles can be developed that interoperate with UPnP™ (Universal Plug and Play) devices and UPnP control points. The specification is based on the UPnP Device Architecture and does not further explain the UPnP specifications. The UPnP specifications are maintained by [1] *Open Connectivity Foundation*.

UPnP™ is a trademark of the UPnP Implementers Corporation.

111.1.1 Essentials

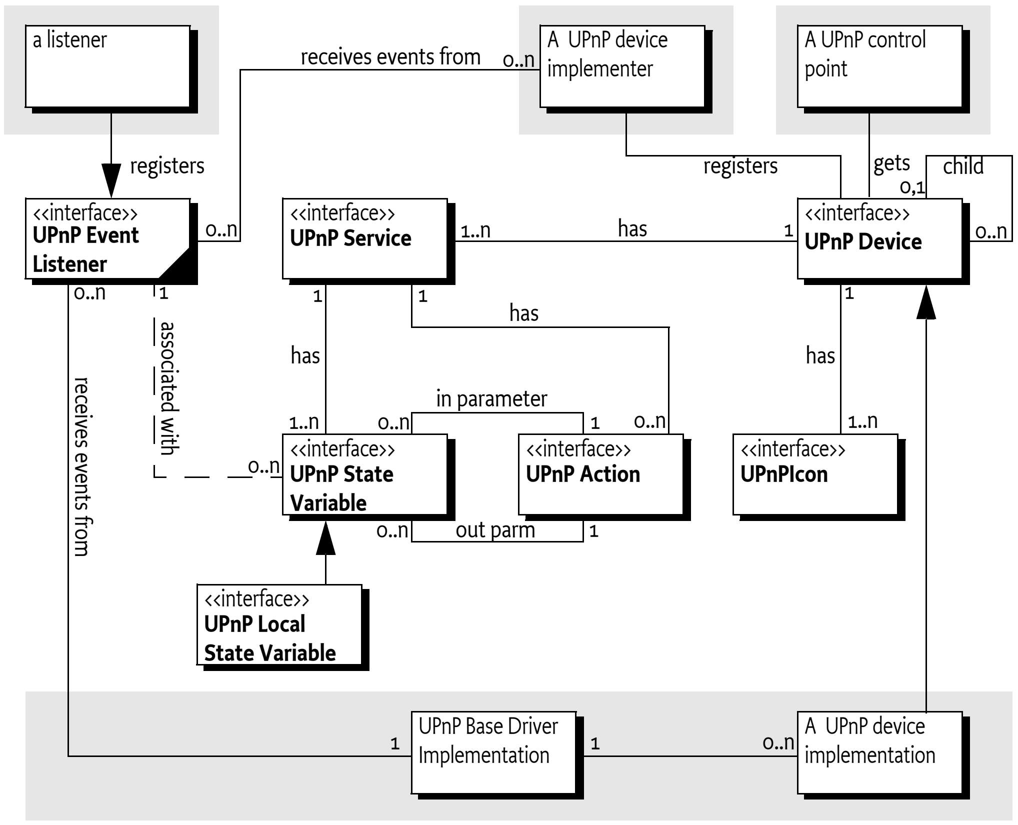
- *Scope* - This specification is limited to device control aspects of the UPnP specifications. Aspects concerning the TCP/IP layer, like DHCP and limited TTL, are not addressed.
- *Transparency* - OSGi services should be made available to networks with UPnP enabled devices in a transparent way.
- *Network Selection* - It must be possible to restrict the use of the UPnP protocols to a selection of the connected networks. For example, in certain cases OSGi services that are UPnP enabled should not be published to the Wide Area Network side of a gateway, nor should UPnP devices be detected on this WAN.
- *Event handling* - Bundles must be able to listen to UPnP events.
- *Export OSGi services as UPnP devices* - Enable bundles that make a service available to UPnP control points.
- *Implement UPnP Control Points* - Enable bundles that control UPnP devices.

111.1.2 Entities

- *UPnP Base Driver* - The bundle that implements the bridge between OSGi and UPnP networks. This entity is not represented as a service.
- *UPnP Root Device* -A physical device can contain one or more root devices. Root devices contain one ore more devices. A root device is modeled with a `UPnPDevice` object, there is no separate interface defined for root devices.
- *UPnP Device* - The representation of a UPnP device. A UPnP device may contain other UPnP devices and UPnP services. This entity is represented by a `UPnPDevice` object. A device can be local (implemented in the Framework) or external (implemented by another device on the net).

- *UPnP Service* -A UPnP device consists of a number of services. A UPnP service has a number of UPnP state variables that can be queried and modified with actions. This concept is represented by a UPnPService object.
- *UPnP Action* - A UPnP service is associated with a number of actions that can be performed on that service and that may modify the UPnP state variables. This entity is represented by a UPnPAction object.
- *UPnP State Variable* - A variable associated with a UPnP service, represented by a UPnPStateVariable object.
- *UPnP Local State Variable* - Extends the UPnPStateVariable interface when the state variable is implemented locally. This interface provides access to the actual value.
- *UPnP Event Listener Service* - A listener to events coming from UPnP devices.
- *UPnP Host* - The machine that hosts the code to run a UPnP device or control point.
- *UPnP Control Point* - A UPnP device that is intended to control UPnP devices over a network. For example, a UPnP remote controller.
- *UPnP Icon* - A representation class for an icon associated with a UPnP device.
- *UPnP Exception* - An exception that delivers errors that were discovered in the UPnP layer.
- *UDN* - Unique Device Name, a name that uniquely identifies the a specific device.

Figure 111.1 UPnP Service Specification class Diagram org.osgi.service.upnp package


a listener
receives events from
0..n
A UPnP device implementer
A UPnP control point
registers
registers
gets
0,1
child
<<interface>> UPnP Event Listener
0..n
<<interface>> UPnP Service
1..n
has
1
<<interface>> UPnP Device
0..n
0..n
1
1
1
has
1
associated with
receives events from
has
has
in parameter
1..n
0..n
1
0..n
1..n
0..n
<<interface>> UPnP State Variable
<<interface>> UPnP Action
<<interface>> UPnPIcon
0..n
out parm
1
<<interface>> UPnP Local State Variable
1
UPnP Base Driver Implementation
1
0..n
A UPnP device implementation

111.1.3 Operation Summary

To make a UPnP service available to UPnP control points on a network, an OSGi service object must be registered under the UPnPDevice interface with the Framework. The UPnP driver bundle must detect these UPnP Device services and must make them available to the network as UPnP devices using the UPnP protocol.

UPnP devices detected on the local network must be detected and automatically registered under the `UPnPDevice` interface with the Framework by the UPnP driver implementation bundle.

A bundle that wants to control UPnP devices, for example to implement a UPnP control point, should track UPnP Device services in the OSGi service registry and control them appropriately. Such bundles should not distinguish between resident or remote UPnP Device services.

111.2 UPnP Specifications

The UPnP DA is intended to be used in a broad range of device from the computing (PCs printers), consumer electronics (DVD, TV, radio), communication (phones) to home automation (lighting control, security) and home appliances (refrigerators, coffee makers) domains.

For example, a UPnP TV might announce its existence on a network by broadcasting a message. A UPnP control point on that network can then discover this TV by listening to those announce messages. The UPnP specifications allow the control point to retrieve information about the user interface of the TV. This information can then be used to allow the end user to control the remote TV from the control point, for example turn it on or change the channels.

The UPnP specification supports the following features:

- *Detect and control a UPnP standardized device.* In this case the control point and the remote device share a priori knowledge about how the device should be controlled. The UPnP Forum intends to define a large number of these standardized devices.
- *Use a user interface description.* A UPnP control point receives enough information about a device and its services to automatically build a user interface for it.
- *Programmatic Control.* A program can directly control a UPnP device without a user interface. This control can be based on detected information about the device or through a priori knowledge of the device type.
- *Allows the user to browse a web page supplied by the device.* This web page contains a user interface for the device that be directly manipulated by the user. However, this option is not well defined in the UPnP Device Architecture specification and is not tested for compliance.

The UPnP Device Architecture specification and the OSGi Framework provide *complementary* functionality. The UPnP Device Architecture specification is a data communication protocol that does not specify where and how programs execute. That choice is made by the implementations. In contrast, the OSGi Framework specifies a (managed) execution point and does not define what protocols or media are supported. The UPnP specification and the OSGi specifications are fully complementary and do not overlap.

From the OSGi perspective, the UPnP specification is a communication protocol that can be implemented by one or more bundles. This specification therefore defines the following:

- How an OSGi bundle can implement a service that is exported to the network via the UPnP protocols.
- How to find and control services that are available on the local network.

The UPnP specifications related to the assignment of IP addresses to new devices on the network or auto-IP self configuration should be handled at the operating system level. Such functions are outside the scope of this specification.

111.2.1 UPnP Base Driver

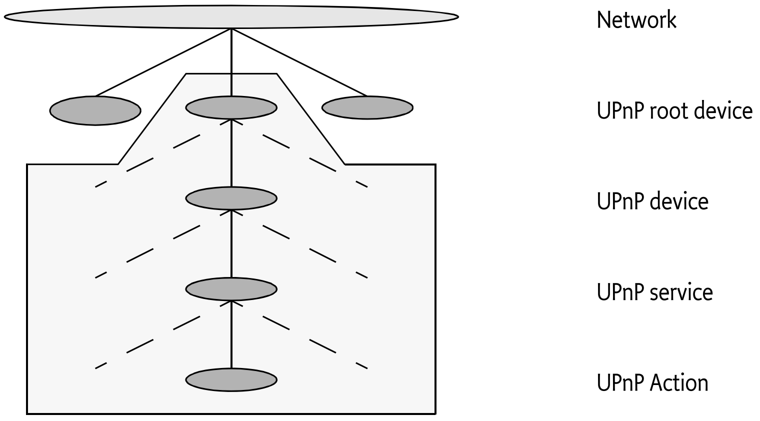
The functionality of the UPnP service is implemented in a UPnP *base driver*. This is a bundle that implements the UPnP protocols and handles the interaction with bundles that use the UPnP devices. A UPnP base driver bundle must provide the following functions:

- Discover UPnP devices on the network and map each discovered device into an OSGi registered UPnP Device service.
- Present UPnP marked services that are registered with the OSGi Framework on one or more networks to be used by other computers.

111.3 UPnP Device

The principle entity of the UPnP specification is the UPnP device. There is a UPnP *root device* that represents a physical appliance, such as a complete TV. The root device contains a number of sub-devices. These might be the tuner, the monitor, and the sound system. Each sub-device is further composed of a number of UPnP services. A UPnP service represents some functional unit in a device. For example, in a TV tuner it can represent the TV channel selector. Figure 111.2 on page 222 illustrates this hierarchy.

Figure 111.2 UPnP device hierarchy



Each UPnP service can be manipulated with a number of UPnP actions. UPnP actions can modify the state of a UPnP state variable that is associated with a service. For example, in a TV there might be a state variable *volume*. There are then actions to set the volume, to increase the volume, and to decrease the volume.

111.3.1 Root Device

The UPnP root device is registered as a UPnP Device service with the Framework, as well as all its sub-devices. Most applications will work with sub-devices, and, as a result, the children of the root device are registered under the `UPnPDevice` interface.

UPnP device properties are defined per sub-device in the UPnP specification. These properties must be registered with the OSGi Framework service registry so they are searchable.

Bundles that want to handle the UPnP device hierarchy can use the registered service properties to find the parent of a device (which is another registered `UPnPDevice`).

The following service registration properties can be used to discover this hierarchy:

- PARENT_UDN - (String) The Universal Device Name (UDN) of the parent device. A root device most not have this property registered. Type is a String object.
- CHILDREN_UDN - (String[]) An array of UDNs of this device's children.

111.3.2 Exported Versus Imported Devices

Both imported (from the network to the OSGi service registry) and exported (from the service registry to the network) UPnPDevice services must have the same representation in the OSGi Framework for identical devices. For example, if an OSGi UPnP Device service is exported as a UPnP device from an OSGi Framework to the network, and it is imported into another OSGi Framework, the object representation should be equal. Application bundles should therefore be able to interact with imported and exported forms of the UPnP device in the same manner.

Imported and exported UPnP devices differ only by two marker properties that can be added to the service registration. One marker, DEVICE_CATEGORY, should typically be set only on imported devices. By not setting DEVICE_CATEGORY on internal UPnP devices, the Device Manager does not try to refine these devices (See the *Device Access Specification* on page 39 for more information about the Device Manager). If the device service does not implement the Device interface and does not have the DEVICE_CATEGORY property set, it is not considered a *device* according to the Device Access Specification.

The other marker, UPNP_EXPORT, should only be set on internally created devices that the bundle developer wants to export. By not setting UPNP_EXPORT on registered UPnP Device services, the UPnP Device service can be used by internally created devices that should not be exported to the network. This allows UPnP devices to be simulated within an OSGi Framework without announcing all of these devices to any networks.

The UPNP_EXPORT service property has no defined type, any value is correct.

111.3.3 Icons

A UPnP device can optionally support an icon. The purpose of this icon is to identify the device on a UPnP control point. UPnP control points can be implemented in large computers like PC's or simple devices like a remote control. However, the graphic requirements for these UPnP devices differ tremendously. The device can, therefore, export a number of icons of different size and depth.

In the UPnP specifications, an icon is represented by a URL that typically refers to the device itself. In this specification, a list of icons is available from the UPnP Device service.

In order to obtain localized icons, the method getIcons(String) can be used to obtain different versions. If the locale specified is a null argument, then the call returns the icons of the default locale of the called device (not the default locale of the UPnP control point).When a bundle wants to access the icon of an imported UPnP device, the UPnP driver gets the data and presents it to the application through an input stream.

A bundle that needs to export a UPnP Device service with one or more icons must provide an implementation of the UPnPIcon interface. This implementation must provide an InputStream object to the actual icon data. The UPnP driver bundle must then register this icon with an HTTP server and include the URL to the icon with the UPnP device data at the appropriate place.

111.4 Device Category

UPnP Device services are devices in the context of the Device Manager. This means that these services need to register with a number of properties to participate in driver refinement. The value for UPnP devices is defined in the UPnPDevice constant DEVICE_CATEGORY. The value is UPnP. The UPnPDevice interface contains a number of constants for matching values. Refer to MATCH_GENERIC for further information.

111.5 UPnPService

A UPnP Device contains a number of UPnPService objects. UPnPService objects combine zero or more actions and one or more state variables.

111.5.1 State Variables

The UPnPStateVariable interface encapsulates the properties of a UPnP state variable. In addition to the properties defined by the UPnP specification, a state variable is also mapped to a Java data type. The Java data type is used when an event is generated for this state variable and when an action is performed containing arguments related to this state variable. There must be a strict correspondence between the UPnP data type and the Java data type so that bundles using a particular UPnP device profile can predict the precise Java data type.

The function QueryStateVariable defined in the UPnP specification has been deprecated and is therefore not implemented. It is recommended to use the UPnP event mechanism to track UPnP state variables.

Additionally, a UPnPStateVariable object can also implement the UPnPLocalStateVariable interface if the device is implemented locally. That is, the device is not imported from the network. The UPnPLocalStateVariable interface provides a getCurrentValue() method that provides direct access to the actual value of the state variable.

111.6 Working With a UPnP Device

The UPnP driver must register all discovered UPnP devices in the local networks. These devices are registered under a UPnPDevice interface with the OSGi Framework.

Using a remote UPnP device thus involves tracking UPnP Device services in the OSGi service registry. The following code illustrates how this can be done. The sample Controller class extends the ServiceTracker class so that it can track all UPnP Device services and add them to a user interface, such as a remote controller application.

```
class Controller extends ServiceTracker {
    UI ui;

    Controller( BundleContext context ) {
        super( context, UPnPDevice.class.getName(), null );
    }
    public Object addingService( ServiceReference ref ) {
        UPnPDevice dev = (UPnPDevice)super.addingService(ref);
        ui.addDevice( dev );
        return dev;
    }
    public void removedService( ServiceReference ref,
        Object dev ) {
        ui.removeDevice( (UPnPDevice) dev );
    }
    ...
}
```

111.7 Implementing a UPnP Device

OSGi services can also be exported as UPnP devices to the local networks, in a way that is transparent to typical UPnP devices. This allows developers to bridge legacy devices to UPnP networks. A bundle should perform the following to export an OSGi service as a UPnP device:

- Register an UPnP Device service with the registration property UPNP_EXPORT.
- Use the registration property PRESENTATION_URL to provide the presentation page. The service implementer must register its own servlet with the Http Service to serve out this interface. This URL must point to that servlet.

There can be multiple UPnP root devices hosted by one OSGi platform. The relationship between the UPnP devices and the OSGi platform is defined by the PARENT_UDN and CHILDREN_UDN service properties. The bundle registering those device services must make sure these properties are set accordingly.

Devices that are implemented on the OSGi Framework (in contrast with devices that are imported from the network) should use the UPnPLocalStateVariable interface for their state variables instead of the UPnPStateVariable interface. This interface provides programmatic access to the actual value of the state variable as maintained by the device specific code.

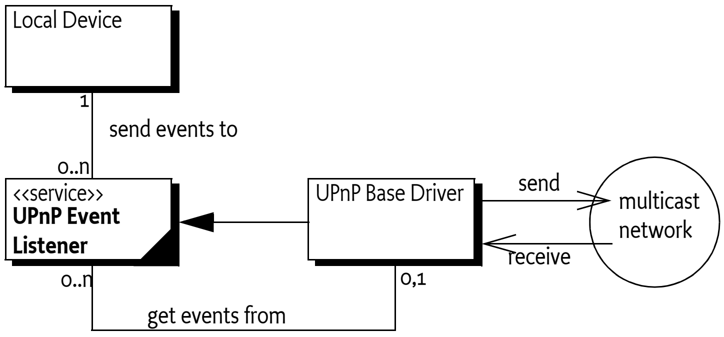
111.8 Event API

There are two distinct event directions for the UPnP Service specification.

- External events from the network must be dispatched to listeners inside the OSGi Frameworks. The UPnP Base driver is responsible for mapping the network events to internal listener events.
- Implementations of UPnP devices must send out events to local listeners as well as cause the transmission of the UPnP network events.

UPnP events are sent using the whiteboard model, in which a bundle interested in receiving the UPnP events registers an object implementing the UPnPEventListener interface. A filter can be set to limit the events for which a bundle is notified. The UPnP Base driver must register a UPnP Event Lister without filter that receives all events.

Figure 111.3 *Event Dispatching for Local and External Devices*



If a service is registered with a property named upnp.filter with the value of an instance of an Filter object, the listener is only notified for matching events (This is a Filter object and not a String object because it allows the InvalidSyntaxException to be thrown in the client and not the UPnP driver bundle).

The filter might refer to any valid combination of the following pseudo properties for event filtering:

- UPnPDevice.UDN - (UPnP.device.UDN/String) Only events generated by services contained in the specific device are delivered. For example: (UPnP.device.UDN=uuid:Upnp-TVEmulator-1_0-1234567890001)
- UPnPDevice.TYPE - (UPnP.device.type/String or String[]) Only events generated by services contained in a device of the given type are delivered. For example: (UPnP.device.type=urn:schemas-upnp-org:device:tvdevice:1)
- UPnPService.ID - (UPnP.service.id/String) Service identity. Only events generated by services matching the given service ID are delivered.
- UPnPService.TYPE - (UPnP.service.type/String or String[]) Only events generated by services of the given type are delivered.

If an event is generated by either a local device or via the base driver for an external device, the notifyUPnPEvent(String,String,Dictionary) method is called on all registered UPnPEventListener services for which the optional filter matches for that event. If no filter is specified, all events must be delivered. If the filter does not match, the UPnP Driver must not call the UPnP Event Listener service. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 8*.

One or multiple events are passed as parameters to the notifyUPnPEvent(String,String,Dictionary) method. The Dictionary object holds a pair of UpnPStateVariable objects that triggered the event and an Object for the new value of the state variable.

111.8.1 Initial Event Delivery

Special care must be taken with the initial subscription to events. According to the UPnP specification, when a client subscribes for notification of events for the first time, the device sends out a number of events for each state variable, indicating the current value of each state variable. This behavior simplifies the synchronization of a device and an event-driven client.

The UPnP Base Driver must mimic this event distribution on behalf of external devices. It must therefore remember the values of the state variables of external devices. A UPnP Device implementation must send out these initial events for each state variable they have a value for.

The UPnP Base Driver must have stored the last event from the device and retransmit the value over the multicast network. The UPnP Driver must register an event listener without any filter for this purpose.

The call to the listener's notification method must be done asynchronously.

111.9 UPnP Events and Event Admin service

UPnP events must be delivered asynchronously to the Event Admin service by the UPnP implementation, if present. UPnP events have the following topic:

```
org/osgi/service/upnp/UPnPEvent
```

The properties of a UPnP event are the following:

- upnp.deviceId - (String) The identity as defined by UPnPDevice.UDN of the device sending the event.
- upnp.serviceId - (String) The identity of the service sending the events.
- upnp.events - (Dictionary) A Dictionary object containing the new values for the state variables that have changed.

111.10 Localization

All values of the UPnP properties are obtained from the device using the device's default locale. If an application wants to query a set of localized property values, it has to use the method getDescriptions(String). For localized versions of the icons, the method getIcons(String) is to be used.

111.11 Dates and Times

The UPnP specification uses different types for date and time concepts. An overview of these types is given in the following table.

Table 111.1 *Mapping UPnP Date/Time types to Java*

UPnP Type	Class	Example	Value (TZ=CEST=UTC+0200)
date	Date	1985-04-12	Sun April 12 00:00:00 CEST 1985
dateTime	Date	1985-04-12T10:15:30	Sun April 12 10:15:30 CEST 1985
dateTime.tz	Date	1985-04-12T10:15:30+0400	Sun April 12 08:15:30 CEST 1985
time	Long	23:20:50	84.050.000 (ms)
time.tz	Long	23:20:50+0100	1.250.000 (ms)

The UPnP specification points to [2] *XML Schema*. In this standard, [3] *ISO 8601 Date And Time formats* are referenced. The mapping is not completely defined which means that this OSGi UPnP specification defines a complete mapping to Java classes. The UPnP types date, dateTime and dateTime.tz are represented as a Date object. For the date type, the hours, minutes and seconds must all be zero.

The UPnP types time and time.tz are represented as a Long object that represents the number of ms since midnight. If the time wraps to the next day due to a time zone value, then the final value must be truncated modulo 86.400.000.

See also TYPE_DATE.

111.12 UPnP Exception

The UPnP Exception can be thrown when a UPnPAction is invoked. This exception contains information about the different UPnP layers. The following errors are defined:

INVALID_ACTION - (401) No such action could be found.

INVALID_ARGS - (402) Invalid argument.

INVALID_SEQUENCE_NUMBER - (403) Out of synchronization.

INVALID_VARIABLE - (404) State variable not found.

DEVICE_INTERNAL_ERROR - (501) Internal error.

Further errors are categorized as follows:

- *Common Action Errors* - In the range of 600-69, defined by the UPnP Forum Technical Committee.
- *Action Specific Errors* - In the range of 700-799, defined by the UPnP Forum Working Committee.
- *Non-Standard Action Specific Errors* - In the range of 800-899. Defined by vendors.

111.13 Configuration

In order to provide a standardized way to configure a UPnP driver bundle, the Configuration Admin property `upnp.ssdp.address` is defined.

The value is a `String[]` with a list of IP addresses, optionally followed with a colon (':' `\u003A`) and a port number. For example:

```
239.255.255.250:1900
```

Those addresses define the interfaces which the UPnP driver is operating on. If no SSDP address is specified, the default assumed will be 239.255.255.250:1900. If no port is specified, port 1900 is assumed as default.

111.14 Networking considerations

111.14.1 The UPnP Multicasts

The operating system must support multicasting on the selected network device. In certain cases, a multicasting route has to be set in the operating system routing table.

These configurations are highly dependent on the underlying operating system and beyond the scope of this specification.

111.15 Security

The UPnP specification is based on HTTP and uses plain text SOAP (XML) messages to control devices. For this reason, it does not provide any inherent security mechanisms. However, the UPnP specification is based on the exchange of XML files and not code. This means that at least worms and viruses cannot be implemented using the UPnP protocols.

However, a bundle registering a UPnP Device service is represented on the outside network and has the ability to communicate. The same is true for getting a UPnP Device service. It is therefore recommended that `ServicePermission[UPnPDevice|UPnPEventListener, REGISTER|GET]` be used sparingly and only for bundles that are trusted.

111.16 org.osgi.service.upnp

UPnP Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.upnp; version="[1.2,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.upnp; version="[1.2,1.3)"
```

111.16.1 Summary

- `UPnPAction` - A UPnP action.
- `UPnPDevice` - Represents a UPnP device.
- `UPnPEventListener` - UPnP Events are mapped and delivered to applications according to the OSGi whiteboard model.
- `UPnPException` - There are several defined error situations describing UPnP problems while a control point invokes actions to UPnPDevices.
- `UPnPIcon` - A UPnP icon representation.
- `UPnPLocalStateVariable` - A local UPnP state variable which allows the value of the state variable to be queried.
- `UPnPService` - A representation of a UPnP Service.
- `UPnPStateVariable` - The meta-information of a UPnP state variable as declared in the device's service state table (SST).

111.16.2 public interface UPnPAction

A UPnP action. Each UPnP service contains zero or more actions. Each action may have zero or more UPnP state variables as arguments.

111.16.2.1 public String[] getInputArgumentNames()

□ Lists all input arguments for this action.

Each action may have zero or more input arguments.

This method must continue to return the action input argument names after the UPnP action has been removed from the network.

Returns Array of input argument names or `null` if no input arguments.

See Also UPnPStateVariable

111.16.2.2 public String getName()

□ Returns the action name. The action name corresponds to the `name` field in the `actionList` of the service description.

- For standard actions defined by a UPnP Forum working committee, action names must not begin with `X_` nor `A_`.
- For non-standard actions specified by a UPnP vendor and added to a standard service, action names must begin with `X_`.

This method must continue to return the action name after the UPnP action has been removed from the network.

Returns Name of action, must not contain a hyphen character or a hash character

111.16.2.3 public String[] getOutputArgumentNames()

□ List all output arguments for this action.

This method must continue to return the action output argument names after the UPnP action has been removed from the network.

Returns Array of output argument names or `null` if there are no output arguments.

See Also UPnPStateVariable

111.16.2.4 public String getReturnArgumentName()

□ Returns the name of the designated return argument.

One of the output arguments can be flagged as a designated return argument.

This method must continue to return the action return argument name after the UPnP action has been removed from the network.

Returns The name of the designated return argument or null if none is marked.

111.16.2.5 public UPnPStateVariable getStateVariable(String argumentName)

argumentName The name of the UPnP action argument.

□ Finds the state variable associated with an argument name. Helps to resolve the association of state variables with argument names in UPnP actions.

Returns State variable associated with the named argument or null if there is no such argument.

Throws IllegalStateException– if the UPnP action has been removed from the network.

See Also UPnPStateVariable

111.16.2.6 public Dictionary<String, Object> invoke(Dictionary<String, Object> args) throws Exception

args A Dictionary of arguments. Must contain the correct set and type of arguments for this action. May be null if no input arguments exist.

□ Invokes the action. The input and output arguments are both passed as Dictionary objects. Each entry in the Dictionary object has a String object as key representing the argument name and the value is the argument itself. The class of an argument value must be assignable from the class of the associated UPnP state variable. The input argument Dictionary object must contain exactly those arguments listed by getInputArguments method. The output argument Dictionary object will contain exactly those arguments listed by getOutputArguments method.

Returns A Dictionary with the output arguments. null if the action has no output arguments.

Throws UPnPException– A UPnP error has occurred.

IllegalStateException– if the UPnP action has been removed from the network.

Exception– The execution fails for some reason.

See Also UPnPStateVariable

111.16.3 public interface UPnPDevice

Represents a UPnP device. For each UPnP root and embedded device, an object is registered with the framework under the UPnPDevice interface.

The relationship between a root device and its embedded devices can be deduced using the UPnPDevice.CHILDREN_UDN and UPnPDevice.PARENT_UDN service registration properties.

The values of the UPnP property names are defined by the UPnP Forum.

All values of the UPnP properties are obtained from the device using the device's default locale.

If an application wants to query for a set of localized property values, it has to use the method UPnPDevice.getDescriptions(String locale).

111.16.3.1 public static final String CHILDREN_UDN = "UPnP.device.childrenUDN"

The property key that must be set for all devices containing other embedded devices.

The value is an array of UDNs for each of the device's children (String[]). The array contains UDNs for the immediate descendants only.

If an embedded device in turn contains embedded devices, the latter are not included in the array.

The UPnP Specification does not encourage more than two levels of nesting.

The property is not set if the device does not contain embedded devices.

The property is of type String[]. Value is "UPnP.device.childrenUDN"

111.16.3.2 public static final String DEVICE_CATEGORY = "UPnP"

Constant for the value of the service property DEVICE_CATEGORY used for all UPnP devices. Value is "UPnP".

See Also org.osgi.service.device.Constants.DEVICE_CATEGORY

111.16.3.3 public static final String FRIENDLY_NAME = "UPnP.device.friendlyName"

Mandatory property key for a short user friendly version of the device name. The property value holds a String object with the user friendly name of the device. Value is "UPnP.device.friendlyName".

111.16.3.4 public static final String ID = "UPnP.device.UDN"

Property key for the Unique Device ID property. This property is an alias to UPnPDevice.UDN. It is merely provided for reasons of symmetry with the UPnPService.ID property. The value of the property is a String object of the Device UDN. The value of the key is "UPnP.device.UDN".

111.16.3.5 public static final String MANUFACTURER = "UPnP.device.manufacturer"

Mandatory property key for the device manufacturer's property. The property value holds a String representation of the device manufacturer's name. Value is "UPnP.device.manufacturer".

111.16.3.6 public static final String MANUFACTURER_URL = "UPnP.device.manufacturerURL"

Optional property key for a URL to the device manufacturers Web site. The value of the property is a String object representing the URL. Value is "UPnP.device.manufacturerURL".

111.16.3.7 public static final int MATCH_GENERIC = 1

Constant for the UPnP device match scale, indicating a generic match for the device. Value is 1.

111.16.3.8 public static final int MATCH_MANUFACTURER_MODEL = 7

Constant for the UPnP device match scale, indicating a match with the device model. Value is 7.

111.16.3.9 public static final int MATCH_MANUFACTURER_MODEL_REVISION = 15

Constant for the UPnP device match scale, indicating a match with the device revision. Value is 15.

111.16.3.10 public static final int MATCH_MANUFACTURER_MODEL_REVISION_SERIAL = 31

Constant for the UPnP device match scale, indicating a match with the device revision and the serial number. Value is 31.

111.16.3.11 public static final int MATCH_TYPE = 3

Constant for the UPnP device match scale, indicating a match with the device type. Value is 3.

111.16.3.12 public static final String MODEL_DESCRIPTION = "UPnP.device.modelDescription"

Optional (but recommended) property key for a String object with a long description of the device for the end user. The value is "UPnP.device.modelDescription".

111.16.3.13 public static final String MODEL_NAME = "UPnP.device.modelName"

Mandatory property key for the device model name. The property value holds a String object giving more information about the device model. Value is "UPnP.device.modelName".

111.16.3.14 public static final String MODEL_NUMBER = "UPnP.device.modelNumber"

Optional (but recommended) property key for a String class typed property holding the model number of the device. Value is "UPnP.device.modelNumber".

111.16.3.15 public static final String MODEL_URL = "UPnP.device.modelURL"

Optional property key for a String typed property holding a string representing the URL to the Web site for this model. Value is "UPnP.device.modelURL".

111.16.3.16 public static final String PARENT_UDN = "UPnP.device.parentUDN"

The property key that must be set for all embedded devices. It contains the UDN of the parent device. The property is not set for root devices. The value is "UPnP.device.parentUDN".

111.16.3.17 public static final String PRESENTATION_URL = "UPnP.presentationURL"

Optional (but recommended) property key for a String typed property holding a string representing the URL to a device representation Web page. Value is "UPnP.presentationURL".

111.16.3.18 public static final String SERIAL_NUMBER = "UPnP.device.serialNumber"

Optional (but recommended) property key for a String typed property holding the serial number of the device. Value is "UPnP.device.serialNumber".

111.16.3.19 public static final String TYPE = "UPnP.device.type"

Property key for the UPnP Device Type property. Some standard property values are defined by the Universal Plug and Play Forum. The type string also includes a version number as defined in the UPnP specification. This property must be set.

For standard devices defined by a UPnP Forum working committee, this must consist of the following components in the given order separated by colons:

- urn
- schemas-upnp-org
- device
- a device type suffix
- an integer device version

For non-standard devices specified by UPnP vendors following components must be specified in the given order separated by colons:

- urn
- an ICANN domain name owned by the vendor
- device
- a device type suffix
- an integer device version

To allow for backward compatibility the UPnP driver must automatically generate additional Device Type property entries for smaller versions than the current one. If for example a device announces its type as version 3, then properties for versions 2 and 1 must be automatically generated.

In the case of exporting a UPnPDevice, the highest available version must be announced on the network.

Syntax Example: `urn:schemas-upnp-org:device:deviceType:v`

The value is "UPnP.device.type".

111.16.3.20 public static final String UDN = "UPnP.device.UDN"

Property key for the Unique Device Name (UDN) property. It is the unique identifier of an instance of a UPnPDevice. The value of the property is a String object of the Device UDN. Value of the key is "UPnP.device.UDN". This property must be set.

111.16.3.21 public static final String UPC = "UPnP.device.UPC"

Optional property key for a String typed property holding the Universal Product Code (UPC) of the device. Value is "UPnP.device.UPC".

111.16.3.22 public static final String UPNP_EXPORT = "UPnP.export"

The UPnP.export service property is a hint that marks a device to be picked up and exported by the UPnP Service. Imported devices do not have this property set. The registered property requires no value.

The UPNP_EXPORT string is "UPnP.export".

111.16.3.23 public Dictionary<String, Object> getDescriptions(String locale)

locale A language tag as defined by RFC 1766 and maintained by ISO 639. Examples include "de", "en" or "en-US". The default locale of the device is specified by passing a null argument.

□ Get a set of localized UPnP properties. The UPnP specification allows a device to present different device properties based on the client's locale. The properties used to register the UPnPDevice service in the OSGi registry are based on the device's default locale. To obtain a localized set of the properties, an application can use this method.

Not all properties might be available in all locales. This method does **not** substitute missing properties with their default locale versions.

This method must continue to return the properties after the UPnP device has been removed from the network.

Returns Dictionary mapping property name Strings to property value Strings

111.16.3.24 public UPnPIcon[] getIcons(String locale)

locale A language tag as defined by RFC 1766 and maintained by ISO 639. Examples include "de", "en" or "en-US". The default locale of the device is specified by passing a null argument.

□ Lists all icons for this device in a given locale. The UPnP specification allows a device to present different icons based on the client's locale.

Returns Array of icons or null if no icons are available.

Throws IllegalStateException– if the UPnP device has been removed from the network.

111.16.3.25 public UPnPService getService(String serviceId)

serviceId The service id

□ Locates a specific service by its service id.

Returns The requested service or null if not found.

Throws IllegalStateException– if the UPnP device has been removed from the network.

111.16.3.26 public UPnPService[] getServices()

□ Lists all services provided by this device.

Returns Array of services or null if no services are available.

Throws IllegalStateException– if the UPnP device has been removed from the network.

111.16.4 public interface UPnPEventListener

UPnP Events are mapped and delivered to applications according to the OSGi whiteboard model. An application that wishes to be notified of events generated by a particular UPnP Device registers a service extending this interface.

The notification call from the UPnP Service to any UPnPEventListener object must be done asynchronous with respect to the originator (in a separate thread).

Upon registration of the UPnP Event Listener service with the Framework, the service is notified for each variable which it listens for with an initial event containing the current value of the variable. Subsequent notifications only happen on changes of the value of the variable.

A UPnP Event Listener service filter the events it receives. This event set is limited using a standard framework filter expression which is specified when the listener service is registered.

The filter is specified in a property named "upnp.filter" and has as a value an object of type org.osgi.framework.Filter.

When the Filter is evaluated, the following keywords are recognized as defined as literal constants in the UPnPDevice class.

The valid subset of properties for the registration of UPnP Event Listener services are:

- UPnPDevice.TYPE-- Which type of device to listen for events.
- UPnPDevice.ID-- The ID of a specific device to listen for events.
- UPnPService.TYPE-- The type of a specific service to listen for events.
- UPnPService.ID-- The ID of a specific service to listen for events.

111.16.4.1 public static final String UPNP_FILTER = "upnp.filter"

Key for a service property having a value that is an object of type org.osgi.framework.Filter and that is used to limit received events.

111.16.4.2 public void notifyUPnPEvent(String deviceId, String serviceId, Dictionary<String, Object> events)

deviceId ID of the device sending the events

serviceId ID of the service sending the events

events Dictionary object containing the new values for the state variables that have changed.

□ Callback method that is invoked for received events. The events are collected in a Dictionary object. Each entry has a String key representing the event name (= state variable name) and the new value of the state variable. The class of the value object must match the class specified by the UPnP State Variable associated with the event. This method must be called asynchronously

111.16.5 public class UPnPException extends Exception

There are several defined error situations describing UPnP problems while a control point invokes actions to UPnPDevices.

Since 1.1

111.16.5.1 public static final int DEVICE_INTERNAL_ERROR = 501

The invoked action failed during execution.

111.16.5.2 public static final int INVALID_ACTION = 401

No Action found by that name at this service.

111.16.5.3 public static final int INVALID_ARGS = 402

Not enough arguments, too many arguments with a specific name, or one of more of the arguments are of the wrong type.

111.16.5.4 public static final int INVALID_SEQUENCE_NUMBER = 403

The different end-points are no longer in synchronization.

111.16.5.5 public static final int INVALID_VARIABLE = 404

Refers to a non existing variable.

111.16.5.6 public UPnPException(int errorCode, String errorDescription)

errorCode error code which defined by UPnP Device Architecture V1.0.

errorDescription error description which explain the type of problem.

□ This constructor creates a `UPnPException` on the specified error code and error description.

111.16.5.7 public UPnPException(int errorCode, String errorDescription, Throwable errorCause)

errorCode error code which defined by UPnP Device Architecture V1.0.

errorDescription error description which explain the type of the problem.

errorCause cause of that `UPnPException`.

□ This constructor creates a `UPnPException` on the specified error code, error description and error cause.

Since 1.2

111.16.5.8 public int getUPnPError_Code()

□ Returns the UPnPError Code occurred by UPnPDevices during invocation.

Returns The UPnPErrorCode defined by a UPnP Forum working committee or specified by a UPnP vendor.

Deprecated As of 1.2. Replaced by getUPnPErrorCode().

111.16.5.9 public int getUPnPErrorCode()

□ Returns the UPnP Error Code occurred by UPnPDevices during invocation.

Returns The UPnPErrorCode defined by a UPnP Forum working committee or specified by a UPnP vendor.

Since 1.2

111.16.6 public interface UPnPIcon

A UPnP icon representation. Each UPnP device can contain zero or more icons.

111.16.6.1 public int getDepth()

□ Returns the color depth of the icon in bits.

This method must continue to return the icon depth after the UPnP device has been removed from the network.

Returns The color depth in bits. If the actual color depth of the icon is unknown, -1 is returned.

111.16.6.2 public int getHeight()

□ Returns the height of the icon in pixels. If the actual height of the icon is unknown, -1 is returned.

This method must continue to return the icon height after the UPnP device has been removed from the network.

Returns The height in pixels, or -1 if unknown.

111.16.6.3 public InputStream getInputStream() throws IOException

□ Returns an `InputStream` object for the icon data. The `InputStream` object provides a way for a client to read the actual icon graphics data. The number of bytes available from this `InputStream` object can be determined via the `getSize()` method. The format of the data encoded can be determined by the MIME type available via the `getMimeType()` method.

Returns An InputStream to read the icon graphics data from.

Throws IOException– If the InputStream cannot be returned.

IllegalStateException– if the UPnP device has been removed from the network.

See Also UPnPIcon.getMimeType()

111.16.6.4 public String getMimeType()

□ Returns the MIME type of the icon. This method returns the format in which the icon graphics, read from the InputStream object obtained by the getInputStream() method, is encoded.

The format of the returned string is in accordance to RFC2046. A list of valid MIME types is maintained by the IANA [http://www.iana.org/assignments/media-types/].

Typical values returned include: "image/jpeg" or "image/gif"

This method must continue to return the icon MIME type after the UPnP device has been removed from the network.

Returns The MIME type of the encoded icon.

111.16.6.5 public int getSize()

□ Returns the size of the icon in bytes. This method returns the number of bytes of the icon available to read from the InputStream object obtained by the getInputStream() method. If the actual size can not be determined, -1 is returned.

Returns The icon size in bytes, or -1 if the size is unknown.

Throws IllegalStateException– if the UPnP device has been removed from the network.

111.16.6.6 public int getWidth()

□ Returns the width of the icon in pixels. If the actual width of the icon is unknown, -1 is returned.

This method must continue to return the icon width after the UPnP device has been removed from the network.

Returns The width in pixels, or -1 if unknown.

111.16.7 public interface UPnPLocalStateVariable extends UPnPStateVariable

A local UPnP state variable which allows the value of the state variable to be queried.

Since 1.1

111.16.7.1 public Object getCurrentValue()

□ This method will keep the current values of UPnPStateVariables of a UPnPDevice whenever UPnPStateVariable's value is changed , this method must be called.

Returns Object current value of UPnPStateVariable. If the current value is initialized with the default value defined UPnP service description.

Throws IllegalStateException– If the UPnP state variable has been removed.

111.16.8 public interface UPnPService

A representation of a UPnP Service. Each UPnP device contains zero or more services. The UPnP description for a service defines actions, their arguments, and event characteristics.

111.16.8.1 public static final String ID = "UPnP.service.id"

Property key for the optional service id. The service id property is used when registering UPnP Device services or UPnP Event Listener services. The value of the property contains a String array

(String[]) of service ids. A UPnP Device service can thus announce what service ids it contains. A UPnP Event Listener service can announce for what UPnP service ids it wants notifications. A service id does **not** have to be universally unique. It must be unique only within a device. A null value is a wildcard, matching **all** services. The value is "UPnP.service.id".

111.16.8.2 public static final String TYPE = "UPnP.service.type"

Property key for the optional service type uri. The service type property is used when registering UPnP Device services and UPnP Event Listener services. The property contains a String array (String[]) of service types. A UPnP Device service can thus announce what types of services it contains. A UPnP Event Listener service can announce for what type of UPnP services it wants notifications. The service version is encoded in the type string as specified in the UPnP specification. A null value is a wildcard, matching **all** service types. Value is "UPnP.service.type".

See Also UPnPService.getType()

111.16.8.3 public UPnPAction getAction(String name)

name Name of action. Must not contain hyphen or hash characters. Should be < 32 characters.

□ Locates a specific action by name. Looks up an action by its name.

Returns The requested action or null if no action is found.

Throws IllegalStateException– if the UPnP service has been removed from the network.

111.16.8.4 public UPnPAction[] getActions()

□ Lists all actions provided by this service.

Returns Array of actions (UPnPAction[])or null if no actions are defined for this service.

Throws IllegalStateException– if the UPnP service has been removed from the network.

111.16.8.5 public String getId()

□ Returns the serviceId field in the UPnP service description.

For standard services defined by a UPnP Forum working committee, the serviceId must contain the following components in the indicated order:

- urn:upnp-org:serviceId:
- service ID suffix

Example: urn:upnp-org:serviceId:serviceID.

Note that upnp-org is used instead of schemas-upnp-org in this example because an XML schema is not defined for each serviceId.

For non-standard services specified by UPnP vendors, the serviceId must contain the following components in the indicated order:

- urn:
- ICANN domain name owned by the vendor
- :serviceId:
- service ID suffix

Example: urn:domain-name:serviceId:serviceID.

This method must continue to return the service id after the UPnP service has been removed from the network.

Returns The service ID suffix defined by a UPnP Forum working committee or specified by a UPnP vendor. Must be <= 64 characters. Single URI.

111.16.8.6 public UPnPStateVariable getStateVariable(String name)

name Name of the State Variable

□ Gets a UPnPStateVariable objects provided by this service by name

Returns State variable or null if no such state variable exists for this service.

Throws IllegalStateException– if the UPnP service has been removed from the network.

111.16.8.7 public UPnPStateVariable[] getStateVariables()

□ Lists all UPnPStateVariable objects provided by this service.

Returns Array of state variables or null if none are defined for this service.

Throws IllegalStateException– if the UPnP service has been removed from the network.

111.16.8.8 public String getType()

□ Returns the serviceType field in the UPnP service description.

For standard services defined by a UPnP Forum working committee, the serviceType must contain the following components in the indicated order:

- urn:schemas-upnp-org:service:
- service type suffix:
- integer service version

Example: urn:schemas-upnp-org:service:serviceType:v.

For non-standard services specified by UPnP vendors, the serviceType must contain the following components in the indicated order:

- urn:
- ICANN domain name owned by the vendor
- :service:
- service type suffix:
- integer service version

Example: urn:domain-name:service:serviceType:v.

This method must continue to return the service type after the UPnP service has been removed from the network.

Returns The service type suffix defined by a UPnP Forum working committee or specified by a UPnP vendor. Must be <= 64 characters, not including the version suffix and separating colon. Single URI.

111.16.8.9 public String getVersion()

□ Returns the version suffix encoded in the serviceType field in the UPnP service description.

This method must continue to return the service version after the UPnP service has been removed from the network.

Returns The integer service version defined by a UPnP Forum working committee or specified by a UPnP vendor.

111.16.9 public interface UPnPStateVariable

The meta-information of a UPnP state variable as declared in the device's service state table (SST).

Method calls to interact with a device (e.g. UPnPAction.invoke(...);) use this class to encapsulate meta information about the input and output arguments.

The actual values of the arguments are passed as Java objects. The mapping of types from UPnP data types to Java data types is described with the field definitions.

111.16.9.1 public static final String TYPE_BIN_BASE64 = "bin.base64"

MIME-style Base64 encoded binary BLOB.

Takes 3 Bytes, splits them into 4 parts, and maps each 6 bit piece to an octet. (3 octets are encoded as 4.) No limit on size.

Mapped to `byte[]` object. The Java byte array will hold the decoded content of the BLOB.

111.16.9.2 public static final String TYPE_BIN_HEX = "bin.hex"

Hexadecimal digits representing octets.

Treats each nibble as a hex digit and encodes as a separate Byte. (1 octet is encoded as 2.) No limit on size.

Mapped to `byte[]` object. The Java byte array will hold the decoded content of the BLOB.

111.16.9.3 public static final String TYPE_BOOLEAN = "boolean"

True or false.

Mapped to `Boolean` object.

111.16.9.4 public static final String TYPE_CHAR = "char"

Unicode string.

One character long.

Mapped to `Character` object.

111.16.9.5 public static final String TYPE_DATE = "date"

A calendar date.

Date in a subset of ISO 8601 format without time data.

See http://www.w3.org/TR/ xmlschema-2/#date [http://www.w3.org/TR/xmlschema-2/#date].

Mapped to `java.util.Date` object. Always 00:00 hours.

111.16.9.6 public static final String TYPE_DATETIME = "dateTime"

A specific instant of time.

Date in ISO 8601 format with optional time but no time zone.

See http://www.w3.org /TR/xmlschema-2/#dateTime [http://www.w3.org/TR/xmlschema-2/#dateTime].

Mapped to `java.util.Date` object using default time zone.

111.16.9.7 public static final String TYPE_DATETIME_TZ = "dateTime.tz"

A specific instant of time.

Date in ISO 8601 format with optional time and optional time zone.

See http://www.w3.org /TR/xmlschema-2/#dateTime [http://www.w3.org/TR/xmlschema-2/#dateTime].

Mapped to `java.util.Date` object adjusted to default time zone.

111.16.9.8 public static final String TYPE_FIXED_14_4 = "fixed.14.4"

Same as r8 but no more than 14 digits to the left of the decimal point and no more than 4 to the right.

Mapped to Double object.

111.16.9.9 public static final String TYPE_FLOAT = "float"

Floating-point number.

Mantissa (left of the decimal) and/or exponent may have a leading sign. Mantissa and/or exponent may have leading zeros. Decimal character in mantissa is a period, i.e., whole digits in mantissa separated from fractional digits by period. Mantissa separated from exponent by E. (No currency symbol.) (No grouping of digits in the mantissa, e.g., no commas.)

Mapped to Float object.

111.16.9.10 public static final String TYPE_I1 = "i1"

1 Byte int.

Mapped to Integer object.

111.16.9.11 public static final String TYPE_I2 = "i2"

2 Byte int.

Mapped to Integer object.

111.16.9.12 public static final String TYPE_I4 = "i4"

4 Byte int.

Must be between -2147483648 and 2147483647

Mapped to Integer object.

111.16.9.13 public static final String TYPE_INT = "int"

Integer number.

Mapped to Integer object.

111.16.9.14 public static final String TYPE_NUMBER = "number"

Same as r8.

Mapped to Double object.

111.16.9.15 public static final String TYPE_R4 = "r4"

4 Byte float.

Same format as float. Must be between 3.40282347E+38 to 1.17549435E-38.

Mapped to Float object.

111.16.9.16 public static final String TYPE_R8 = "r8"

8 Byte float.

Same format as float. Must be between -1.79769313486232E308 and -4.94065645841247E-324 for negative values, and between 4.94065645841247E-324 and 1.79769313486232E308 for positive values, i.e., IEEE 64-bit (8-Byte) double.

Mapped to Double object.

111.16.9.17 public static final String TYPE_STRING = "string"

Unicode string.

No limit on length.

Mapped to String object.

111.16.9.18 public static final String TYPE_TIME = "time"

An instant of time that recurs every day.

Time in a subset of ISO 8601 format with no date and no time zone.

See http://www.w3.org /TR/xmlschema-2/#time [http://www.w3.org/TR/xmlschema-2/#dateTime].

Mapped to Long. Converted to milliseconds since midnight.

111.16.9.19 public static final String TYPE_TIME_TZ = "time.tz"

An instant of time that recurs every day.

Time in a subset of ISO 8601 format with optional time zone but no date.

See http://www.w3.org /TR/xmlschema-2/#time [http://www.w3.org/TR/xmlschema-2/#dateTime].

Mapped to Long object. Converted to milliseconds since midnight and adjusted to default time zone, wrapping at 0 and 24*60*60*1000.

111.16.9.20 public static final String TYPE_UI1 = "ui1"

Unsigned 1 Byte int.

Mapped to an Integer object.

111.16.9.21 public static final String TYPE_UI2 = "ui2"

Unsigned 2 Byte int.

Mapped to Integer object.

111.16.9.22 public static final String TYPE_UI4 = "ui4"

Unsigned 4 Byte int.

Mapped to Long object.

111.16.9.23 public static final String TYPE_URI = "uri"

Universal Resource Identifier.

Mapped to String object.

111.16.9.24 public static final String TYPE_UUID = "uuid"

Universally Unique ID.

Hexadecimal digits representing octets. Optional embedded hyphens are ignored.

Mapped to String object.

111.16.9.25 public String[] getAllowedValues()

□ Returns the allowed values, if defined. Allowed values can be defined only for String types.

This method must continue to return the state variable allowed values after the UPnP state variable has been removed from the network.

Returns The allowed values or null if not defined. Should be less than 32 characters.

111.16.9.26 public Object getDefaultValue()

□ Returns the default value, if defined.

This method must continue to return the state variable default value after the UPnP state variable has been removed from the network.

Returns The default value or `null` if not defined. The type of the returned object can be determined by `getJavaDataType`.

111.16.9.27 public Class<?> getJavaDataType()

□ Returns the Java class associated with the UPnP data type of this state variable.

Mapping between the UPnP data types and Java classes is performed according to the schema mentioned above.

Integer	ui1, ui2, i1, i2, i4, int
Long	ui4, time, time.tz
Float	r4, float
Double	r8, number, fixed.14.4
Character	char
String	string, uri, uuid
Date	date, dateTime, dateTime.tz
Boolean	boolean
byte[]	bin.base64, bin.hex

This method must continue to return the state variable java type after the UPnP state variable has been removed from the network.

Returns A class object corresponding to the Java type of this argument.

111.16.9.28 public Number getMaximum()

□ Returns the maximum value, if defined. Maximum values can only be defined for numeric types.

This method must continue to return the state variable maximum value after the UPnP state variable has been removed from the network.

Returns The maximum value or `null` if not defined.

111.16.9.29 public Number getMinimum()

□ Returns the minimum value, if defined. Minimum values can only be defined for numeric types.

This method must continue to return the state variable minimum value after the UPnP state variable has been removed from the network.

Returns The minimum value or `null` if not defined.

111.16.9.30 public String getName()

□ Returns the variable name.

- All standard variables defined by a UPnP Forum working committee must not begin with `X_` nor `A_`.
- All non-standard variables specified by a UPnP vendor and added to a standard service must begin with `X_`.

This method must continue to return the state variable name after the UPnP state variable has been removed from the network.

Returns Name of state variable. Must not contain a hyphen character nor a hash character. Should be < 32 characters.

111.16.9.31 public Number getStep()

□ Returns the size of an increment operation, if defined. Step sizes can be defined only for numeric types.

This method must continue to return the step size after the UPnP state variable has been removed from the network.

Returns The increment size or null if not defined.

111.16.9.32 public String getUPnPDataType()

□ Returns the UPnP type of this state variable. Valid types are defined as constants.

This method must continue to return the state variable UPnP data type after the UPnP state variable has been removed from the network.

Returns The UPnP data type of this state variable, as defined in above constants.

111.16.9.33 public boolean sendsEvents()

□ Tells if this StateVariable can be used as an event source. If the StateVariable is eventable, an event listener service can be registered to be notified when changes to the variable appear.

This method must continue to return the correct value after the UPnP state variable has been removed from the network.

Returns `true` if the `StateVariable` generates events, `false` otherwise.

111.17 References

[1] *Open Connectivity Foundation*
https://openconnectivity.org/developer/specifications/upnp-resources/upnp/

[2] *XML Schema*
https://www.w3.org/TR/xmlschema-2

[3] *ISO 8601 Date And Time formats*
https://www.iso.org/iso-8601-date-and-time-format.html

112 Declarative Services Specification

Version 1.5

112.1 Introduction

The OSGi Framework contains a procedural service model which provides a publish/find/bind model for using *services*. This model is elegant and powerful, it enables the building of applications out of bundles that communicate and collaborate using these services.

This specification addresses some of the complications that arise when the OSGi service model is used for larger systems and wider deployments, such as:

- *Startup Time* - The procedural service model requires a bundle to actively register and acquire its services. This is normally done at startup time, requiring all present bundles to be initialized with a Bundle Activator. In larger systems, this quickly results in unacceptably long startup times.
- *Memory Footprint* - A service registered with the Framework implies that the implementation, and related classes and objects, are loaded in memory. If the service is never used, this memory is unnecessarily occupied. The creation of a class loader may therefore cause significant overhead.
- *Complexity* - Service can come and go at any time. This dynamic behavior makes the service programming model more complex than more traditional models. This complexity negatively influences the adoption of the OSGi service model as well as the robustness and reliability of applications because these applications do not always handle the dynamicity correctly.

The *service component* model uses a declarative model for publishing, finding and binding to OSGi services. This model simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies. This minimizes the amount of code a programmer has to write; it also allows service components to be loaded only when they are needed. As a result, bundles need not provide a `BundleActivator` class to collaborate with others through the service registry.

From a system perspective, the service component model means reduced startup time and potentially a reduction of the memory footprint. From a programmer's point of view the service component model provides a simplified programming model.

The Service Component model makes use of concepts described in [1] *Automating Service Dependency Management in a Service-Oriented Component Model.*

112.1.1 Essentials

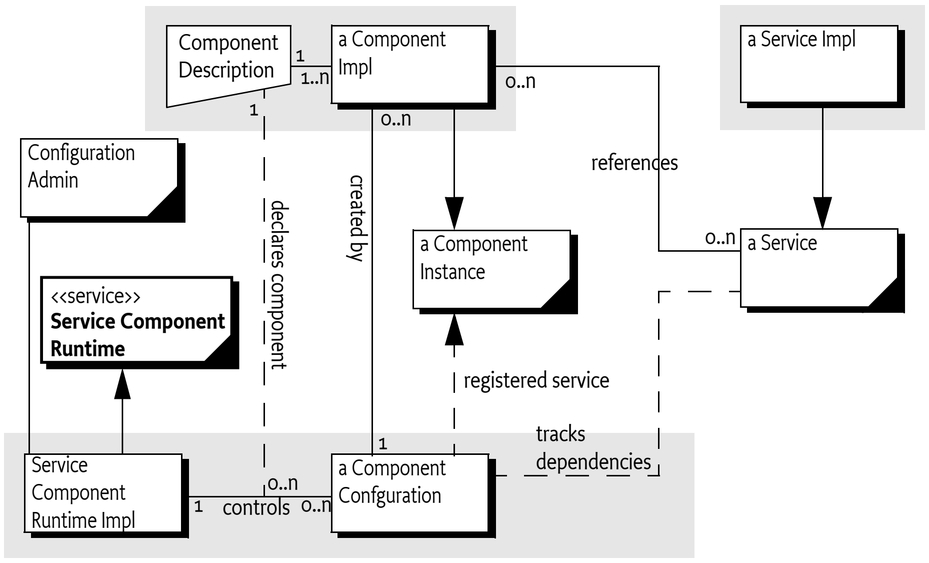
- *Backward Compatibility* - The service component model must operate seamlessly with the existing service model.
- *Size Constraints* - The service component model must not require memory and performance intensive subsystems. The model must also be applicable on resource constrained devices.
- *Delayed Activation* - The service component model must allow delayed activation of a service component. Delayed activation allows for delayed class loading and object creation until needed, thereby reducing the overall memory footprint.
- *Simplicity* - The programming model for using declarative services must be very simple and not require the programmer to learn a complicated API or XML sub-language.

- *Dependency Injection* - The programming model for using declarative services supports three types of dependency injection: method injection, field injection, and constructor injection.
- *Reactive* - It must be possible to react to changes in the external dependencies with different policies.
- *Annotations* - Annotations must be provided that can leverage the type information to create the XML descriptor.
- *Introspection* - It must be possible to introspect the service components.

112.1.2 Entities

- *Service Component* - A service component contains a description that is interpreted at run time to create and dispose objects depending on the availability of other services, the need for such an object, and available configuration data. Such objects can optionally provide a service. This specification also uses the generic term *component* to refer to a service component.
- *Service Component Runtime (SCR)* - The actor that manages the components and their life cycle and allows introspection of the components.
- *Component Description* - The declaration of a service component. It is contained within an XML document in a bundle.
- *Component Properties* - A set of properties which can be specified by the component description, Configuration Admin service and from the component factory.
- *Component Property Type* - A user defined annotation type which defines component properties and is implemented by SCR to provide type safe access to the defined component properties.
- *Component Configuration* - A component configuration represents a component description parameterized by component properties. It is the entity that tracks the component dependencies and manages a component instance. An activated component configuration has a component context.
- *Component Instance* - An instance of the component implementation class. A component instance is created when a component configuration is activated and discarded when the component configuration is deactivated. A component instance is associated with exactly one component configuration.
- *Delayed Component* - A component whose component configurations are activated when their service is requested.
- *Immediate Component* - A component whose component configurations are activated immediately upon becoming satisfied.
- *Factory Component* - A component whose component configurations are created and activated through the component's component factory.
- *Reference* - A specified dependency of a component on a set of target services.
- *Target Services* - The set of services that is defined by the reference interface and target property filter.
- *Bound Services* - The set of target services that are bound to a component configuration.
- *Event methods* - The bind, updated, and unbind methods associated with a Reference.

Figure 112.1 *Service Component Runtime, org.osgi.service.component package*



112.1.3 Synopsis

The Service Component Runtime reads component descriptions from started bundles. These descriptions are in the form of XML documents which define a set of components for a bundle. A component can refer to a number of services that must be available before a component configuration becomes satisfied. These dependencies are defined in the descriptions and the specific target services can be influenced by configuration information in the Configuration Admin service. After a component configuration becomes satisfied, a number of different scenarios can take place depending on the component type:

- *Immediate Component* - The component configuration of an immediate component must be activated immediately after becoming satisfied. Immediate components may provide a service.
- *Delayed Component* - When a component configuration of a delayed component becomes satisfied, SCR will register the service specified by the `service` element without activating the component configuration. If this service is requested, SCR must activate the component configuration creating an instance of the component implementation class that will be returned as the service object. If the `scope` attribute of the `service` element is `bundle`, then, for each distinct bundle that requests the service object, a different component configuration is created and activated and a new instance of the component implementation class is returned as the service object. If the `scope` attribute of the `service` element is `prototype`, then, for each distinct request for the service object, such as via `ServiceObjects`, a different component configuration is created and activated and a new instance of the component implementation class is returned as the service object.
- *Factory Component* - If a component's description specifies the `factory` attribute of the `component` element, SCR will register a Component Factory service. This service allows client bundles to create and activate multiple component configurations and dispose of them. If the component's description also specifies a `service` element, then as each component configuration is activated, SCR will register it as a service.

112.1.4 Readers

- *Architects* - The chapter, *Components* on page 248, gives a comprehensive introduction to the capabilities of the component model. It explains the model with a number of examples. The section about *Component Life Cycle* on page 276 provides some deeper insight in the life cycle of components.

- *Service Programmers* - Service programmers should read *Components* on page 248. This chapter should suffice for the most common cases. For the more advanced possibilities, they should consult *Component Description* on page 264 for the details of the XML grammar for component descriptions.
- *Deployers* - Deployers should consult *Deployment* on page 287.

112.2 Components

A component is a normal Java class contained within a bundle. The distinguishing aspect of a component is that it is *declared* in an XML document. Component configurations are activated and deactivated under the full control of SCR. SCR bases its decisions on the information in the component's description. This information consists of basic component information like the name and type, optional services that are implemented by the component, and *references*. References are dependencies that the component has on other services.

SCR must *activate* a component configuration when the component is enabled and the component configuration is satisfied and a component configuration is needed. During the life time of a component configuration, SCR can notify the component of changes in its bound references.

SCR will *deactivate* a previously activated component configuration when the component becomes disabled, the component configuration becomes unsatisfied, or the component configuration is no longer needed.

If an activated component configuration's configuration properties change, SCR must either notify the component configuration of the change, if the component description specifies a method to be notified of such changes, or deactivate the component configuration and then attempt to reactivate the component configuration using the new configuration information.

112.2.1 Declaring a Component

A component requires the following artifacts in the bundle:

- An XML document that contains the component description.
- The `Service-Component` manifest header which names the XML documents that contain the component descriptions.
- An implementation class that is specified in the component description.

The elements in the component's description are defined in *Component Description* on page 264.
The XML grammar for the component declaration is defined by the XML Schema, see *Component Description Schema* on page 303.

112.2.2 Immediate Component

An *immediate component* is activated as soon as its dependencies are satisfied. If an immediate component has no dependencies, it is activated immediately. A component is an immediate component if it is not a factory component and either does not specify a service or specifies a service and the `immediate` attribute of the `component` element set to `true`. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration.

For example, the bundle entry `/OSGI-INF/activator.xml` contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.activator"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.Activator"/>
```

```
</scr:component>
```

The manifest header Service-Component must also be specified in the bundle manifest. For example:

```
Service-Component: OSGI-INF/activator.xml
```

An example class for this component could look like:

```
public class Activator {
    public Activator() {...}
    private void activate(BundleContext context) {...}
    private void deactivate() {...}
}
```

This example component is virtually identical to a Bundle Activator. It has no references to other services so it will be satisfied immediately. It publishes no service so SCR will activate a component configuration immediately.

The activate method is called when SCR activates the component configuration and the deactivate method is called when SCR deactivates the component configuration. If the activate method throws an Exception, then the component configuration is not activated and will be discarded.

112.2.3 Delayed Component

A *delayed component* specifies a service, is not specified to be a factory component and does not have the immediate attribute of the component element set to true. If a delayed component configuration is satisfied, SCR must register the component configuration as a service in the service registry but the activation of the component configuration is delayed until the registered service is requested. The registered service of a delayed component looks like a normal registered service but does not incur the overhead of an ordinarily registered service that require a service's bundle to be initialized to register the service.

For example, a bundle needs to see events of a specific topic. The Event Admin uses the white board pattern, receiving the events is therefore as simple as registering a Event Handler service. The example XML for the delayed component looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.handler"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.HandlerImpl"/>
    <property name="event.topics">some/topic</property>
    <service>
        <provide interface="org.osgi.service.event.EventHandler"/>
    </service>
</scr:component>
```

The associated component class looks like:

```
public class HandlerImpl implements EventHandler{
    public void handleEvent(Event evt ) {
        ...
  }
}
```

The component configuration will only be activated once the Event Admin service requires the service because it has an event to deliver on the topic to which the component subscribed.

112.2.4 Factory Component

Certain software patterns require the creation of component configurations on demand. For example, a component could represent an application that can be launched multiple times and each application instance can then quit independently. Such a pattern requires a factory that creates the instances. This pattern is supported with a *factory component*. A factory component is used if the factory attribute of the component element is set to a *factory identifier*. This identifier can be used by a bundle to associate the factory with externally defined information.

SCR must register a Component Factory service on behalf of the component as soon as the component factory is satisfied. The service properties for the Component Factory service are the *factory properties* as specified by the factory-property and factory-properties elements of the component description. See *Factory Property and Factory Properties Elements* on page 274. The service properties of the Component Factory service must not include the component properties. SCR always adds the following factory properties, which cannot be overridden:

- component.name - The name of the component.
- component.factory - The factory identifier.

New configurations of the component can be created and activated by calling the newInstance method on this Component Factory service. The newInstance(Dictionary) method has a Dictionary object as a parameter. This Dictionary object is merged with the component properties as described in *Component Properties* on page 285. If the component specifies a service, then the service is registered after the created component configuration is satisfied with the component properties. Then the component configuration is activated.

For example, a component can provide a connection to a USB device. Such a connection should normally not be shared and should be created each time such a service is needed. The component description to implement this pattern looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.factory"
    factory="usb.connection"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.USBConnectionImpl"/>
</scr:component>
```

The component class looks like:

```
public class USBConnectionImpl {
    private void activate(Map<String, ?> properties) {
     ...
    }
}
```

A factory component can be associated with a service. In that case, such a service is registered for each component configuration. For example, the previous example could provide a USB Connection service.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.factory"
    factory="usb.connection"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.USBConnectionImpl"/>
    <service>
        <provide interface="com.acme.usb.USBConnection"/>
    </service>
```

```
</scr:component>
```

The associated component class looks like:

```
public class USBConnectionImpl implements USBConnection {
    private void activate(Map<String, ?> properties) {...}
    public void connect() { ... }
    ...
    public void close() { ... }
}
```

A new service will be registered each time a new component configuration is created and activated with the newInstance method. This allows a bundle other than the one creating the component configuration to utilize the service. If the component configuration is deactivated, the service must be unregistered.

112.3 References to Services

Most bundles will require access to other services from the service registry. The dynamics of the service registry require care and attention of the programmer because referenced services, once acquired, could be unregistered at any moment. The component model simplifies the handling of these service dependencies significantly.

The services that are selected by a reference are called the *target services*. These are the services selected by the BundleContext.getServiceReferences method where the first argument is the reference's interface and the second argument is the reference's target property, which must be a valid filter.

A component configuration becomes *satisfied* when each specified reference is satisfied. A reference is *satisfied* if it specifies optional cardinality or when the number of target services is equal to or more than the minimum cardinality of the reference. An activated component configuration that becomes *unsatisfied* must be deactivated.

During the activation of a component configuration, SCR must bind some or all of the target services of a reference to the component configuration. Any target service that is bound to the component configuration is called a *bound* service. See *Bound Services* on page 280.

112.3.1 Accessing Services

A component instance must be able to use the services that are referenced by the component configuration, that is, the bound services of the references. The following techniques are available for a component instance to acquire these bound services:

- *Method injection* - SCR calls a method on the component instance when a service becomes bound, when a service becomes unbound, or when its properties are updated. These methods are the bind, updated, and unbind methods specified by the reference. Method injection is useful if the component needs to be notified of changes to the bound services for a dynamic reference.
- *Field injection* - SCR modifies a field in the component instance when a service becomes bound, when a service becomes unbound, or when its properties are updated.
- *Constructor injection* - When SCR activates a component instance, the component instance must be constructed and constructor injection occurs. Bound services and activation objects can be parameters to the constructor.
- *Lookup strategy* - A component instance can use one of the locateService methods of its ComponentContext to locate a bound service. These methods take the name of the reference as a parameter. If the reference has a dynamic policy, it is important to not store returned service objects but look them up every time they are needed.

A component may use multiple strategies to access the bound services of a reference.

112.3.2 Method Injection

When using method injection, SCR must call the component instance at the appropriate time. SCR must call on the following events:

- `bind` - The bind method, if specified, is called to bind a new service to the component that matches the selection criteria. If the `policy` is `dynamic` then the bind method of a replacement service can be called before its corresponding unbind method.
- `updated` - The updated method, if specified, is called when the service properties of a bound services are modified and the resulting properties do not cause the service to become unbound because it is no longer selected by the target property.
- `unbind` - The unbind method, if specified, is called when SCR needs to unbind the service.

Each event is associated with an *event method*.

An event method can take one or more parameters. Each parameter must be of one of the following types:

- `<service-type>` - The bound service object.
- `ServiceReference` - A Service Reference for the bound service. This Service Reference may later be passed to the `locateService(String,ServiceReference)` method to obtain the actual service object. This approach is useful when the service properties need to be examined before accessing the service object. It also allows for the delayed activation of bound services when using method injection.
- `ComponentServiceObjects` - A Component Service Objects for the bound service. This Component Service Objects can be used to obtain the actual service object or objects. This approach is useful when the referenced service has prototype service scope and the component instance needs multiple service objects for the service.
- `Map` - An unmodifiable Map containing the service properties of the bound service. This Map must additionally implement `Comparable` with the `compareTo` method comparing service property maps using an ordering which is the same as the natural ordering of `ServiceReference`s as specified in `ServiceReference.compareTo`.

A suitable method is selected using the following priority. If the type specified by the reference's `interface` attribute is `org.osgi.service.component.AnyService`, then the parameter type must be `java.lang.Object` to match.

1. The method takes a single parameter and the type of the parameter is `org.osgi.framework.ServiceReference`. This method will receive a Service Reference for the bound service.
2. The method takes a single parameter and the type of the parameter is `ComponentServiceObjects`. This method will receive a Component Service Objects for the bound service.
3. The method takes a single parameter and the type of the parameter is the type specified by the reference's `interface` attribute. This method will receive the bound service object.
4. The method takes a single parameter and the type of the parameter is assignable from the type specified by the reference's `interface` attribute. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call. This method will receive the bound service object.
5. The method takes a single parameter and the type of the parameter is `java.util.Map`. This method will receive an unmodifiable Map containing the service properties of the bound service.
6. The method takes two or more parameters and the types of the parameters must be one of: the type specified by the reference's `interface` attribute, a type assignable from the type specified by the reference's `interface` attribute, `org.osgi.framework.ServiceReference`, `ComponentServiceObjects`, or `java.util.Map`. If multiple methods match this rule, this implies the method

name is overloaded and SCR may choose any of the methods to call. In the case where the type specified by the reference's interface attribute is org.osgi.framework.ServiceReference, ComponentServiceObjects, or java.util.Map, the first parameter of that type will receive the bound service object. If selected event method has more than one parameter of that type, the remaining parameters of that type will receive a Service Reference for the bound service, a Service Objects for the bound service, or an unmodifiable Map containing the service properties of the bound service.

When searching for an event method to call, SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 298. If no suitable method is located, SCR must log an error message with the Log Service, if present, and there will be no bind, updated, or unbind notification.

The bind and unbind methods must be called once for each bound service. This implies that if the reference has multiple cardinality, then the methods may be called multiple times. The updated method can be called multiple times per service.

In the following examples, a component requires the Logger Factory service. The first example uses the lookup strategy. The reference is declared without any bind, updated, and unbind methods:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.LogLookupImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LoggerFactory"/>
</scr:component>
```

The component implementation class must now lookup the service. This looks like:

```
public class LogLookupImpl {
    private void activate(ComponentContext ctxt) {
        LoggerFactory lf = ctxt.locateService("LOG");
        lf.getLogger(LogLookupImpl.class).info("Hello Components!");
    }
}
```

Alternatively, the component could use method injection and ask to be notified with the Logger Factory service by declaring bind, updated, and unbind methods.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.LogEventImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LoggerFactory"
        bind="setLog"
        updated="updatedLog"
        unbind="unsetLog"
    />
</scr:component>
```

The component implementation class looks like:

```
public class LogEventImpl {
    LoggerFactory lf;
    Integer    level;
    void setLog( LoggerFactory l, Map<String,?> ref ) {
        lf = l;
        updatedLog(l, ref);
    }
    void updatedLog( LoggerFactory l, Map<String,?> ref) {
        level = (Integer) ref.get("level");
    }
    void unsetLog( LoggerFactory l ) { lf = null; }
    private void activate() {
        lf.getLogger(LogEventImpl.class).info("Hello Components!");
    }
}
```

Event methods can be declared private in the component class but are only looked up in the inheritance chain when they are protected, public, or have default access. See *Locating Component Methods and Fields* on page 298.

112.3.3 Field Injection

When using field injection, SCR must modify fields in the component instance at the appropriate time. SCR must modify the fields on the following events:

- bind - The field is modified to bind a new service to the component that matches the selection criteria.
- updated - For certain field types, the field is modified when the service properties of a bound services are modified and the resulting properties do not cause the service to become unbound because it is no longer selected by the target property.
- unbind - The field is modified when SCR needs to unbind the service.

For a reference with unary cardinality, a field must be of one of the following types:

- <service-type> - The bound service object. The type of the field can be the actual service type or it can be a type that is assignable from the actual service type. If the type specified by the reference's interface attribute is org.osgi.service.component.AnyService, then the actual service type is considered to be java.lang.Object.
- ServiceReference - A Service Reference for the bound service. This Service Reference may later be passed to the locateService(String,ServiceReference) method to obtain the actual service object. This approach is useful when the service properties need to be examined before accessing the service object. It also allows for the delayed activation of bound services when using field injection.
- ComponentServiceObjects - A Component Service Objects for the bound service. This Component Service Objects can be used to obtain the actual service object or objects. This approach is useful when the referenced service has prototype service scope and the component instance needs multiple service objects for the service.
- Map - An unmodifiable Map containing the service properties of the bound service. This Map must additionally implement Comparable with the compareTo method comparing service property maps using an ordering which is the same as the natural ordering of ServiceReferences as specified in ServiceReference.compareTo.
- Map.Entry - An unmodifiable Map.Entry whose key is an unmodifiable Map containing the service properties of the bound service, as above, and whose value is the bound service object. This Map.Entry must additionally implement Comparable with the compareTo method comparing the service property map key using an ordering which is the same as the natural ordering of ServiceReferences as specified in ServiceReference.compareTo.
- Optional - An Optional holding one of the above types. The type of object held in the Optional is specified by the field-collection-type attribute in the component description. The Optional must be empty for an optional reference with no bound service.

If the actual service type is one of ServiceReference, ComponentServiceObjects, Map, Map.Entry, or Optional the field will be set to the service object rather than the object about the service.

For a reference with multiple cardinality, a field must be a collection of one of the following types:

- Collection
- List
- A subtype of Collection - This type can only be used for dynamic references using the update reference field option. The component instance must initialize the field to a collection object in its constructor.

The type of objects set in the collection or the type of object held in the Optional is specified by the field-collection-type attribute in the component description:

- service - The bound service object. This is the default field collection type.
- reference - A Service Reference for the bound service.
- serviceobjects - A Component Service Objects for the bound service.
- properties - An unmodifiable Map containing the service properties of the bound service. This Map must implement Comparable, as above.
- tuple - An unmodifiable Map.Entry whose key is an unmodifiable Map containing the service properties of the bound service, as above, and whose value is the bound service object. This Map.Entry must implement Comparable, as above.

Only instance fields of the field types above are supported. If a referenced field is declared with the static modifier or has a type other than one of the above, SCR must log an error message with the Log Service, if present, and the field must not be modified. SCR must locate a suitable field as specified in *Locating Component Methods and Fields* on page 298. If no suitable field is located, SCR must log an error message with the Log Service, if present, and no field will not be modified for the reference. If the bound service cannot be assigned to the field because the type is unassignable, SCR must log an error message containing the exception with the Log Service, if present, and no field will not be modified for the reference.

Care must be taken by the component implementation regarding the field. SCR has no way to know if the component implementation itself may alter the field value. The component implementation should not alter the field value and allow SCR to manage it. SCR must treat the field as if the component implementation does not alter the field value so SCR may retain its own copy of the value set in the field.

In the following examples, a component requires the Logger Factory service.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.LogEventImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LoggerFactory"
        field="lf"
    />
</scr:component>
```

The component implementation class looks like:

```
public class LogEventImpl {
    LoggerFactory lf;
    private void activate() {
        lf.getLogger(LogEventImpl.class).info("Hello Components!");
    }
}
```

Fields can be declared private in the component class but are only looked up in the inheritance chain when they are protected, public, or have default access. See *Locating Component Methods and Fields* on page 298.

112.3.4 Constructor Injection

When using constructor injection, SCR must construct the component instance using the appropriate constructor passing activation objects and bound services as parameters. Since a component instance is only constructed once, constructor parameters for references must be for static references.

A suitable constructor is selected using the following steps:

1. If the constructor is not public, then the constructor must not be considered.
2. If the constructor has a parameter count that does not match the value of the `init` attribute in the `component` element, then the constructor must not be considered. If the value of the `init` attribute is 0, the default value, then the public no-parameter constructor must be used.
3. For the constructor parameters associated with a reference, that is, there is a reference with a `parameter` attribute whose value matches the zero-based parameter number of the constructor parameter, if the parameter type is not one of the types supported for field injection for a static reference, then the constructor must not be considered. See *Field Injection* on page 254 for information on types supported for field injection.
4. For the constructor parameters not associated with a reference, if the parameter type is not assignable from one of the activation object types, then the constructor must not be considered. See *Activation Objects* on page 280 for information on activation object types.
5. If only a single constructor remains, this constructor must be used to construct the component instance.
6. If more than one constructor remains, this implies the constructor is overloaded and SCR may choose any of the remaining constructors to construct the component instance.

When searching for the constructor to call, SCR must use reflection on the implementation class. If no suitable constructor is located, SCR must log an error message with the Log Service, if present, and the component configuration is not activated.

If the constructor throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the component configuration is not activated. If the bound service cannot be assigned to the constructor parameter because the type is unassignable, SCR must log an error message containing the exception with the Log Service, if present, and the component configuration is not activated.

If the constructor parameter is associated with a reference having cardinality of 0..1 and there is no bound service for the reference, then the value `null` will be supplied as the constructor parameter.

In the following examples, a component requires the Logger Factory service.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen" init="1"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.LogEventImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LoggerFactory"
        parameter="0"
    />
</scr:component>
```

The component implementation class looks like:

```
public class LogEventImpl {
    public LogEventImpl(LoggerFactory lf) {
        lf.getLogger(LogEventImpl.class).info("Hello Components!");
    }
}
```

112.3.5 Reference Cardinality

A component implementation is always written with a certain *cardinality* for each reference in mind. The cardinality represents two important concepts:

- *Multiplicity* - Does the component implementation assume a single service or does it explicitly handle multiple services? For example, when a component uses the Logger Factory service, it only needs to bind to one Logger Factory service to function correctly. Alternatively, when the Configuration Admin uses the Configuration Listener services it needs to bind to all target services present in the service registry to dispatch its events correctly.
- *Optionality* - Can the component function without any bound service present? Some components can still perform useful tasks even when no service is available; other components must bind to at least one service before they can be useful. For example, the Configuration Admin in the previous example must still provide its functionality even if there are no Configuration Listener services present. Alternatively, an application that registers a Servlet with the Http Service has little to do when the Http Service is not present, it should therefore use a reference with a mandatory cardinality.

The cardinality is expressed with the following syntax:

```
cardinality  ::= optionality '..' multiplicity
optionality  ::= '0' | '1'
multiplicity ::= '1' | 'n'
```

The cardinality for a reference can be specified as one of four choices:

- `0..1` - Optional and unary.
- `1..1` - Mandatory and unary (Default) .
- `0..n` - Optional and multiple.
- `1..n` - Mandatory and multiple.

The *minimum cardinality* is specified by the optionality part of the cardinality. This is either `0` or `1`. A minimum cardinality property can be used to raise the minimum cardinality of a reference from this initial value. For example, a `0..n` cardinality in the component description can be raised into a `3..n` cardinality at runtime by setting the minimum cardinality property for the reference to `3`. This would typically be done by a deployer setting the minimum cardinality property in a configuration for the component. The minimum cardinality for a unary cardinality cannot exceed `1`. See *Minimum Cardinality Property* on page 287 for more information.

A reference is *satisfied* if the number of target services is equal to or more than the minimum cardinality. The `multiplicity` is irrelevant for the satisfaction of the reference. The `multiplicity` only specifies if the component implementation is written to handle being bound to multiple services (`n`) or requires SCR to select and bind to a single service (`1`).

When a satisfied component configuration is activated, there must be at most one bound service for each reference with a unary cardinality and at least as many bound services as the minimum cardinality for each reference. If the cardinality constraints cannot be maintained after a component configuration is activated, that is the reference becomes unsatisfied, the component configuration must be deactivated. If the reference has a unary cardinality and there is more than one target service for the reference, then the bound service must be the target service whose ServiceReference is first in the ranking order as specified in ServiceReference.compareTo.

In the following example, a component wants to register a resource with all Http Services that are available. Such a scenario has the cardinality of 0..n. The code must be prepared to handle multiple calls to the bind method for each Http Service in such a case. In this example, the code uses the registerResources method to register a directory for external access.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.HttpResourceImpl"/>
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
        cardinality="0..n"
        bind="setPage"
        unbind="unsetPage"
    />
</scr:component>

public class HttpResourceImpl {
    private void setPage(HttpService http) {
        http.registerResources("/scr", "scr", null );
    }
    private void unsetPage(HttpService http) {
        http.unregister("/scr");
    }
}
```

112.3.6 Reference Scope

A component implementation must be written to understand the service scope of referenced services. The *reference scope* defines whether the component expects the bundle to be exposed to a single service object for a bound service or to potentially multiple services objects. The following reference scopes are available:

- *bundle* - For all references to a given bound service, all activated component instances within a bundle must use the same service object. That is, for a given bound service, all component instances within a bundle will be using the same service object. This is the default reference scope.
- *prototype* - For all references to a given bound service, each activated component instance may use a single, distinct service object. That is, for a given bound service, each component instance may use a distinct service object but within a component instance all references to the bound service will use the same service object.
- *prototype_required* - For all references to a given bound service, each activated component instance must use a single, distinct service object. That is, for a given bound service, each component instance will use a distinct service object but within a component instance all references to the bound service will use the same service object.

For a bound service of a reference with bundle reference scope, SCR must get the service object from the OSGi Framework's service registry using the getService method on the component's Bundle Context. If the service object for a bound service has been obtained and the service becomes unbound, SCR must unget the service object using the ungetService method on the component's Bundle Context and discard all references to the service object. This ensures that the bundle will only be exposed to a single instance of the service object at any given time.

For a bound service of a reference with prototype or prototype_required reference scope, SCR must use a Service Objects object obtained from the OSGi Framework's service registry using the component's Bundle Context to get any service objects. If service objects for a bound service have been obtained and the service becomes unbound, SCR must unget any unreleased service objects using the Service Objects object obtained from the OSGi Framework's service registry using the component's Bundle Context. This means that if a component instance used a Component Service Objects object to obtain service objects, SCR must track those service objects so that when the service becomes unbound, SCR can unget any unreleased service objects.

Additionally, for a reference with prototype_required reference scope, only services registered with prototype service scope can be considered as target services. This ensures that each component instance can be exposed to a single, distinct instance of the service object. Using prototype_required reference scope effectively adds `service.scope=prototype` to the target property for the reference. A service that does not use prototype service scope cannot be used as a bound service for a reference with prototype_required reference scope since the service cannot provide a distinct service object for each component instance.

112.3.7 Reference Policy

Once all the references of a component are satisfied, a component configuration can be activated and therefore bound to target services. However, the dynamic nature of the OSGi service registry makes it likely that services are registered, modified and unregistered after target services are bound. These changes in the service registry could make one or more bound services no longer a target service thereby making obsolete any object references that the component has to these service objects. Components therefore must specify a *policy* how to handle these changes in the set of bound services. A *policy-option* can further refine how changes affect bound services.

112.3.7.1 Static Reference Policy

The *static policy* is the most simple policy and is the default policy. A reference with a static policy is called a *static reference*. A component instance never sees any of the dynamics of the static reference. The bind method is called and/or the field is set before the component instance is activated. Static references can also be used for parameters for constructor injection. Component configurations are deactivated before any bound service for the static reference becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and the replacement service is bound to the new component instance.

If the `policy-option` is `reluctant` then the registration of an additional target service for a reference must not result in deactivating and reactivating a component configuration. If the `policy-option` is `greedy` then the component configuration must be reactivated when new applicable services become available. See Table 112.1 on page 260.

If a static reference specifies an updated method and the bound service's properties change, SCR must call the updated method.

The static policy can be very expensive if it depends on services that frequently unregister and reregister or if the cost of activating and deactivating a component configuration is high. Static policy is usually also not applicable if the cardinality specifies multiple bound services.

112.3.7.2 Dynamic Reference Policy

The *dynamic policy* is slightly more complex since the component implementation must properly handle changes in the set of bound services that can occur on any thread at any time after the component instance is created. A reference with a dynamic policy is called a *dynamic reference*. With a dynamic reference, SCR can change the set of bound services without deactivating a component configuration. If the component uses method injection to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind, updated, and unbind methods.

If the `policy-option` is `reluctant` then a bound reference is not rebound even if a more suitable service becomes available for a 1..1 or 0..1 reference. If the `policy-option` is `greedy` then the component must be unbound and rebound for that reference. See Table 112.1 on page 260.

The previous example with the registering of a resource directory used a static policy. This implied that the component configurations are deactivated when there is a change in the bound set of Http Services. The code in the example can be seen to easily handle the dynamics of Http Services that come and go. The component description can therefore be updated to:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.HttpResourceImpl"/>
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
        cardinality="0..n"
        policy="dynamic"
        bind="setPage"
        unbind="unsetPage"
    />
</scr:component>
```

The code is identical to the previous example.

112.3.8 Reference Policy Option

The reference policy option defines how eager the reference is to rebind when a new, potentially a higher ranking, target service becomes available. The reference policy option can have the following values:

- `reluctant` - Minimize rebinding and reactivating. This is the default reference policy option.
- `greedy` - Maximize the use of the best service by deactivating static references or rebinding dynamic references.

Table 112.1 defines the actions that are taken when a *better* target service becomes available. In this context, better is when the reference is not bound or when the new target service has a higher ranking than the bound service.

Table 112.1 Action taken for policy-option when a new or higher ranking service becomes available

Cardinality	**static reluctant**	**static greedy**	**dynamic reluctant**	**dynamic greedy**
0..1	Ignore	Reactivate to bind the better target service.	If no service is bound, bind to new target service. Otherwise, ignore new target service.	If no service is bound, bind to better target service. Otherwise, unbind the bound service and bind the better target service.
1..1	Ignore	Reactivate to bind the better target service.	Ignore	Unbind the bound service, then bind the new service.
0..n	Ignore	Reactivate	Bind new target service	Bind new target service
1..n	Ignore	Reactivate	Bind new target service	Bind new target service

112.3.9 Reference Field Option

For a reference using field injection, the reference field option defines how SCR must manage the field value. The reference field option can have the following values:

- `replace` - SCR must set the field value. Any field value set by the constructor of the component instance is overwritten. This is the default reference field option.

- `update` - SCR must update the collection set in the field. This collection can be set by the constructor of the component instance. This reference field option can only be used for a dynamic reference with multiple cardinality.

For a static reference, the replace option must be used.

For a dynamic reference, the choice of reference field option is influenced by the cardinality of the reference. For unary cardinality, the replace option must be used. For multiple cardinality, either the replace or update option can be used.

If the update option is used when not permitted, SCR must log an error message with the Log Service, if present, and the field must not be modified.

112.3.9.1 Replace Field Option

If the field is declared with the `final` modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.

For a static reference, SCR must set the field value before the component instance is activated and must not change the field while the component is active. This means there is a *happens-before* relationship between setting the field and activating the component instance, so the active component can safely read the field.

For a dynamic reference, the field must be declared with the `volatile` modifier so that field value changes made by SCR are visible to other threads. If the field is not declared with the `volatile` modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.

For a reference with unary cardinality, SCR must set the field value with initial bound service, if any, before the component instance is activated. If the reference has optional cardinality and there is no bound service, SCR must set the field value to `null`. If the reference is dynamic, when there is a new bound service or the service properties of the bound service are modified and the field holds service properties, SCR must replace the field value. If the reference has optional cardinality and there is no bound service, SCR must set the field value to `null`.

For a reference with multiple cardinality, the type of the field must be `Collection` or `List`. If the field has a different type, SCR must log an error message with the Log Service, if present, and the field must not be modified. Before the component instance is activated, SCR must set the field value with a new mutable collection that must contain the initial set of bound services sorted using an ordering which is the same as the natural ordering of `ServiceReference`s as specified in `ServiceReference.compareTo`. The collection may be empty if the reference has optional cardinality and there are no bound services. If the reference is dynamic, when there is a change in the set of bound services or the service properties of a bound service are modified and the collection holds service properties, SCR must replace the field value with a new mutable collection that must contain the updated set of bound services sorted using an ordering which is the same as the natural ordering of `ServiceReference`s as specified in `ServiceReference.compareTo`. The new collection may be empty if the reference has optional cardinality and there are no bound services.

112.3.9.2 Update Field Option

The update option can only be used for a dynamic reference with multiple cardinality. The component's constructor can set the field with its choice of collection implementation. In this case, the field can be declared with the `final` modifier. The collection implementation used by the component should use identity rather than `equals` or `hashCode` to manage the elements of the collection. The collection implementation should also be thread-safe since SCR may update the collection from threads different than those used by the component instance.

After constructing the component instance, if the field value is `null`:

- If the type of the field is Collection or List, SCR will set the field value to a new mutable empty collection or list object, respectively. If the field is declared with the final modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.
- Otherwise, SCR must log an error message with the Log Service, if present, and the field must not be modified.

SCR must not change the field value while the component is active and only update the contents of the collection. SCR must update the collection before the component instance is activated by calling Collection.add for each bound service. When there is a change to the set of bound services:

- SCR must call Collection.add for a newly bound service.
- SCR must call Collection.remove for an unbound service.
- If the service properties of a bound service are modified and the collection holds service properties, SCR must call Collection.add for the replacement element followed by Collection.remove for the old element.

The collection may be empty if the reference has optional cardinality and there are no bound services.

112.3.10 Selecting Target Services

The target services for a reference are constrained by the reference's interface name and target property. By specifying a filter in the target property, the programmer and deployer can constrain the set of services that should be part of the target services.

For example, a component wants to track all Component Factory services that have a factory identification of acme.application. The following component description shows how this can be done.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.FactoryTracker"/>
    <reference name="FACTORY"
        interface=
            "org.osgi.service.component.ComponentFactory"
        target="(component.factory=acme.application)"
    />
</scr:component>
```

The filter is manifested as a component property called the *target property*. The target property can also be set by property and properties elements, see *Property and Properties Elements* on page 268. The deployer can also set the target property by establishing a configuration for the component which sets the value of the target property. This allows the deployer to override the target property in the component description. See *Target Property* on page 287 for more information.

112.3.10.1 Any Service Type

A special exception to the selecting of target services is the use of the special interface name org.osgi.service.component.AnyService. This means that a service type is not used to select the target services and just the target property is used to select the target services. When the special interface name of org.osgi.service.component.AnyService is used, a target property must be present to constrain the target services to some subset of all available services. If no target property is present, SCR must log an error message with the Log Service, if present, and the reference cannot be satisfied. Note that there may be performance impacts resulting from a target property matching too broad a set of target services. A target property of (service.id=*) is valid but will match all services in the service registry which will likely be neither very useful nor performant.

The reference member or parameter must have a service type of java.lang.Object to allow SCR to provide any service object.

For example, a component wants to track all Jakarta RESTful Web Services Extension services. Jakarta RESTful Web Services Extension services can be registered under many different service types but must be registered with the service property osgi.jakartars.extension=true. The following component description shows how this can be done.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="jakartars.extension.listener"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.ExtensionListener"/>
    <reference name="EXTENSIONS"
        interface=
            "org.osgi.service.component.AnyService"
        target="(osgi.jakartars.extension=true)"
        cardinality="0..n"
    />
</scr:component>
```

112.3.11 Circular References

It is possible for a set of component descriptions to create a circular dependency. For example, if component A references a service provided by component B and component B references a service provided by component A then a component configuration of one component cannot be satisfied without accessing a partially activated component instance of the other component. SCR must ensure that a component instance is never accessible to another component instance or as a service until it has been fully activated, that is it has returned from its activate method if it has one.

Circular references must be detected by SCR when it attempts to satisfy component configurations and SCR must fail to satisfy the references involved in the cycle and log an error message with the Log Service, if present. However, if one of the references in the cycle has optional cardinality SCR must break the cycle. The reference with the optional cardinality can be satisfied and bound to zero target services. Therefore the cycle is broken and the other references may be satisfied.

112.3.12 Logger Support

SCR provides special support for components having references to the Logger Factory from the Log Service specification. If the reference uses method, field or constructor injection, the referenced service is of type org.osgi.service.log.LoggerFactory, and the type of the parameter or field to receive the service object is of type org.osgi.service.log.Logger or org.osgi.service.log.FormatterLogger, then SCR must obtain the proper type of Logger from the bound Logger Factory service and use the obtained Logger as the service object rather than the service object for the bound Logger Factory service.

To obtain the Logger object to use as the service object, SCR must call the LoggerFactory.getLogger(Bundle bundle, String name, Class loggerType) method passing the bundle declaring the component as the first argument, the fully qualified name of the component implementation class as the second argument, and the type of the parameter or field, org.osgi.service.log.Logger or org.osgi.service.log.FormatterLogger, as the third argument.

For example, the following code will have the logger field set to a Logger object created by SCR from the bound Logger Factory service.

```
@Component
public class MyComponent {
  @Reference(service=LoggerFactory.class)
  private Logger logger;
  @Activate
  void activate(ComponentContext context) {
    logger.trace("activating component id {}",
      context.getProperties().get("component.id"));
  }
}
```

112.3.13 Satisfying Condition

The *Condition Service Specification* in *OSGi Core Release 8* defines *Condition* services representing a particular state at runtime and requires the OSGi Framework to always register the *True Condition* service.

Every component description is defined to have a reference to a *satisfying condition*. The name of this reference must be osgi.ds.satisfying.condition. If a component description does not explicitly declare a reference with the name osgi.ds.satisfying.condition, SCR must augment the component description to add the following implicit reference as the last reference:

```
<reference name="osgi.ds.satisfying.condition"
    interface="org.osgi.service.condition.Condition"
    target="(osgi.condition.id=true)"
    policy="dynamic"
/>
```

The implicit reference is handled in the same manner as any explicit reference in the component description, including handling reference properties. See *Reference Properties* on page 286. In order for a component configuration to be satisfied, the reference for the satisfying condition, like all other references, must be satisfied. The target property for the reference to the satisfying condition, osgi.ds.satisfying.condition.target, can be used to select the satisfying condition. This allows a component configuration to be configured to select which Condition service is the satisfying condition. The implicit reference defaults to the target property value of (osgi.condition.id=true) which targets the True Condition, that is always registered by the OSGi Framework, and thus the implicit reference, using the default target property value, can always be satisfied.

For example, you can use the SatisfyingConditionTarget component property type to set the target property for the reference to the satisfying condition to select a condition indicating that some subsystem needed by the component is ready.

```
@Component
@SatisfyingConditionTarget("(osgi.condition.id=my.subsystem.ready)")
public class MyComponent {
  ...
}
```

A component's satisfying condition can be used as a way to control, via the configuration of component properties, whether a component configuration can be satisfied. That is, it can be a way to "enable" or "disable" a component configuration through its configuration. See *Condition Service Specification* of *OSGi Core Release 8* for additional information on Conditions.

SCR must support the satisfying condition for all components even those with component descriptions in older namespaces.

112.4 Component Description

Component descriptions are defined in XML documents contained in a bundle and any attached fragments.

If SCR detects an error when processing a component description, it must log an error message with the Log Service, if present, and ignore the component description. Errors can include XML parsing errors and ill-formed component descriptions.

112.4.1 Annotations

A number of CLASS retention annotations have been provided to allow tools to construct the component description XML from the Java class files. These annotations will be discussed with the appropriate elements and attributes. Since the naming rules between XML and Java differ, some name changes are necessary.

Multi-word element and attribute names that use a minus sign ('-' \u002D) are changed to camel case. For example, the configuration-pid attribute in the component element is the configurationPid member in the @Component annotation. The annotation class that corresponds to an element starts with an upper case letter. For example the component element is represented by the @Component annotation.

Some elements do not have a corresponding annotation since the annotations can be parameterized by the type information in the Java class. For example, the @Component annotation synthesizes the implement element's class attribute from the type it is applied to.

See *Component Annotations* on page 290 for more information.

112.4.2 Service Component Header

XML documents containing component descriptions must be specified by the Service-Component header in the manifest. The value of the header is a comma separated list of paths to XML entries within the bundle.

```
Service-Component ::= header // See Common Header Syntax in Core
```

The Service-Component header has no architected directives or properties. The header can be left empty.

The last component of each path in the Service-Component header may use wildcards so that Bundle.findEntries can be used to locate the XML document within the bundle and its fragments. For example:

```
Service-Component: OSGI-INF/*.xml
```

A Service-Component manifest header specified in a fragment is ignored by SCR. However, XML documents referenced by a bundle's Service-Component manifest header may be contained in attached fragments.

SCR must process each XML document specified in this header. If an XML document specified by the header cannot be located in the bundle and its attached fragments, SCR must log an error message with the Log Service, if present, and continue.

112.4.3 XML Document

A component description must be in a well-formed XML document, [4] *Extensible Markup Language (XML) 1.0*, stored in a UTF-8 encoded bundle entry. The namespace for component descriptions is:

```
http://www.osgi.org/xmlns/scr/v1.5.0
```

The recommended prefix for this namespace is scr. This prefix is used by examples in this specification. XML documents containing component descriptions may contain a single, root component element or one or more component elements embedded in a larger document. Use of the namespace for component descriptions is mandatory. The attributes and sub-elements of a component element are always unqualified.

If an XML document contains a single, root component element which does not specify a namespace, then the http://www.osgi.org/xmlns/scr/v1.0.0 namespace is assumed. Component descriptions using the http://www.osgi.org/xmlns/scr/v1.0.0 namespace must be treated according to version 1.0 of this specification.

SCR must parse all component elements in the namespace. Elements not in this namespace must be ignored. Ignoring elements that are not recognized allows component descriptions to be embedded in any XML document. For example, an entry can provide additional information about components. These additional elements are parsed by another sub-system.

See *Component Description Schema* on page 303 for component description schema.

112.4.4 Component Element

The component element specifies the component description. The following text defines the structure of the XML grammar using a form that is similar to the normal grammar used in OSGi specifications. In this case the grammar should be mapped to XML elements:

```
<component> ::= (<property> | <properties>)*
                <service>?
                <reference>*
                <implementation>
```

SCR must not require component descriptions to specify the elements in the order listed above and as required by the XML schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of property and properties elements and of reference elements have meaning.

The component element has the attributes and @Component annotations defined in the following table.

Table 112.2 Component Element and Annotations

Attribute	Annotation	Description
name	name	The *name* of a component must be unique within a bundle. The component name is used as a PID to retrieve component properties from the OSGi Configuration Admin service if present, unless a configuration-pid attribute has been defined. See *Deployment* on page 287 for more information. If the component name is used as a PID then it should be unique within the framework. The XML schema allows the use of component names which are not valid PIDs. Care must be taken to use a valid PID for a component name if the component should be configured by the Configuration Admin service. This attribute is optional. The default value of this attribute is the value of the class attribute of the nested implementation element. If multiple component elements in a bundle use the same value for the class attribute of their nested implementation element, then using the default value for this attribute will result in duplicate component names. In this case, this attribute must be specified with a unique value.
enabled	enabled	Controls whether the component is *enabled* when the bundle is started. The default value is true. If enabled is set to false, the component is disabled until the method enableComponent is called on the ComponentContext object. This allows some initialization to be performed by some other component in the bundle before this component can become satisfied. See *Enabled* on page 276.
factory	factory	If set to a non-empty string, it indicates that this component is a *factory component*. SCR must register a Component Factory service for each factory component. See *Factory Component* on page 250.

Attribute	Annotation	Description
immediate	immediate	Controls whether component configurations must be immediately activated after becoming satisfied or whether activation should be delayed. The default value is false if the factory attribute or if the service element is specified and true otherwise. If this attribute is specified, its value must be false if the factory attribute is also specified or must be true unless the service element is also specified.
configuration-policy	configurationPolicy OPTIONAL REQUIRE IGNORE	Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component. • optional - (default) Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present. • require - There must be a corresponding Configuration object for the component configuration to become satisfied. • ignore - Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.
configuration-pid	configurationPid	The configuration PIDs to be used for the component in conjunction with Configuration Admin. Multiple configuration PIDs can be specified by using a whitespace separated list in the attribute. The default value for this attribute is the name of the component. The annotation uses a String[] to specify multiple configuration PIDs. The order in which configuration PIDs are specified must be preserved in the generated component description. The annotation can also use the special configuration PID name "$" to specify the name of the component. This special name must be replaced with the actual name of the component in the generated component description.
activate	Activate	Specifies the name of the method to call when a component configuration is activated. The default value of this attribute is activate. See *Activate Method* on page 281 for more information. The Activate annotation must be applied to at most one method which is to be used as the activate method.
activation-fields	Activate	Specifies the whitespace separated list of the names of the fields to hold activation objects. The fields are set once after the constructor has been called and before calling any other method on the fully constructed component instance such as the activate method. See *Activation Objects* on page 280 for more information. The Activate annotation will use the name of the field to which it is applied as the activation field name.
init	Activate	Specifies the number of arguments of the public constructor to use. The default is 0 which represents the public no-parameter constructor. See *Constructor Injection* on page 256 for more information. The Activate annotation must be applied to at most one constructor which is to be used as the constructor for component instances.
deactivate	Deactivate	Specifies the name of the method to call when a component configuration is deactivated. The default value of this attribute is deactivate. See *Deactivate Method* on page 283 for more information. The Deactivate annotation must be applied to at most one method which is to be used as the deactivate method.

Attribute	Annotation	Description
modified	Modified	Specifies the name of the method to call when the configuration properties for a component configuration is using a Configuration object from the Configuration Admin service and that Configuration object is modified without causing the component configuration to become unsatisfied. If this attribute is not specified, then the component configuration will become unsatisfied if its configuration properties use a Configuration object that is modified in any way. See *Modified Method* on page 282 for more information. The Modified annotation must be applied to at most one method which is to be used as the modified method.

112.4.5 Implementation Element

The implementation element is required and defines the name of the component implementation class. The single attribute is defined in the following table.

Table 112.3 Implementation Element and Annotations

Attribute	Annotation	Description
class	Component	The Java fully qualified name of the implementation class. The component Component annotation will define the implementation element automatically from the type it is applied to.

The class is retrieved with the loadClass method of the component's bundle. The class must have a public constructor with the correct parameter count and types which will be used to construct the component instance.

If the component description specifies a service, the class must implement all interfaces that are provided by the service.

112.4.6 Property and Properties Elements

A component description can define a number of properties. These can be defined inline or from a resource in the bundle. The property and properties elements can occur multiple times and they can be interleaved. This interleaving is relevant because the properties are processed from top to bottom. Later properties override earlier properties that have the same name.

Properties can also be overridden by a Configuration Admin service's Configuration object before they are exposed to the component or used as service properties. This is described in *Component Properties* on page 285 and *Deployment* on page 287.

The property element has the attributes and annotations defined in the following table.

Table 112.4 Property Element and Annotations

Attribute	Annotation	Description
name	Component property	The name of the property.
value		The value of the property. This value is parsed according to the property type. If the value attribute is specified, the body of the element is ignored. If the type of the property is not String, parsing of the value is done by the static valueOf(String) method in the given type. For Character types, the conversion must be handled by Integer.valueOf method, a Character is always represented by its Unicode value.

Attribute	Annotation	Description
type		The type of the property. Defines how to interpret the value. The type must be one of the following Java types: • String (default) • Long • Double • Float • Integer • Byte • Character • Boolean • Short
<body>		If the value attribute is not specified, the body of the property element must contain one or more values. The value of the property is then an array of the specified type. The result will be translated to an array of primitives or Strings. For example, if the type attribute specifies Integer, then the resulting array must be int[]. Values must be placed one per line and blank lines are ignored. Parsing of each value is done by the parse methods in the class identified by the type, after trimming the line of any beginning and ending white space. Each String value is also trimmed of beginning and ending white space.

For example, a component that needs an array of hosts can use the following property definition:

```
<property name="hosts">
        www.acme.com
        backup.acme.com
</property>
```

This property declaration results in the property hosts, with a value of String[] { "www.acme.com", "backup.acme.com" }.

A property can also be set with the property annotation element of Component. This element is an array of strings that must follow the following syntax:

```
property ::= name ( ':' type )? '=' value
```

In this case name, type, and value parts map to the attributes of the property element. If multiple values must be specified then the same name can be repeated multiple times. For example:

```
@Component(property={"foo:Integer=1","foo:Integer=2","foo:Integer=3"})
public class FooImpl {
  ...
}
```

The properties element references an entry in the bundle whose contents conform to a standard [3] *Java Properties File.*

At runtime, SCR reads the entry to obtain the properties and their values. The properties element attributes are defined in the following table.

Table 112.5 *Properties Element and Annotations*

Attribute	Annotation	Description
entry	Component properties	The entry path relative to the root of the bundle

For example, to include vendor identification properties that are stored in the OSGI-INF directory, the following definition could be used:

```
<properties entry="OSGI-INF/vendor.properties"/>
```

The properties annotation element of Component can be used to provide the same information. This element consists of an array of strings where each string defines an entry. The order within the array is the order that must be used for the XML. However, the annotations do not support interleaving of the generated property and properties elements.

For example:

```
@Component(properties="OSGI-INF/vendor.properties")
```

See *Ordering of Generated Component Properties* on page 297 for more information on the ordering of generated properties when using annotations.

112.4.7 Service Element

The service element is optional. It describes the service information to be used when a component configuration is to be registered as a service.

A service element has the following attribute defined in the following table.

Table 112.6 Service Element and Annotations

Attribute	Annotation	Description
scope	Component scope SINGLETON BUNDLE PROTOTYPE	Controls the scope of the provided service. If set to singleton, when the component is registered as a service, it must be registered as a bundle scope service but only a single component configuration must be created and activated and a new instance of the component implementation class of the component must be used for all bundles using the service. If set to bundle, when the component is registered as a service, it must be registered as a bundle scope service and a different component configuration is created and activated and a new instance of the component implementation class must be created for each bundle using the service. If set to prototype, when the component is registered as a service, it must be registered as a prototype scope service and a different component configuration is created and activated and a new instance of the component implementation class must be created for each distinct request for the service, such as via ServiceObjects.

The scope attribute must be singleton if the component is a factory component or an immediate component. This is because SCR is not free to create component configurations as necessary to support non-singleton scoped services. A component description is ill-formed if it specifies that the component is a factory component or an immediate component and scope is not singleton.

The service element must have one or more provide elements that define the service interfaces. The provide element has the attribute defined in the following table.

Table 112.7 Provide Element and Annotations

Attribute	Annotation	Description
interface	Component service	The name of the interface that this service is registered under. This name must be the fully qualified name of a Java class. For example, org.osgi.service.eventadmin.EventHandler. The specified Java class should be an interface rather than a class, however specifying a class is supported. The component implementation class must implement all the specified service interfaces. The Component annotation can specify the provided services, if this element is not specified all directly implemented interfaces on the component's type are defined as service interfaces. Specifying an empty array indicates that no service should be registered.

For example, a component implements an Event Handler service.

```
<service>
    <provide interface=
        "org.osgi.service.eventadmin.EventHandler"/>
</service>
```

This previous example can be generated with the following annotation:

```
@Component
public class Foo implements EventHandler { ... }
```

112.4.8 Reference Element

A *reference* declares a dependency that a component has on a set of target services. A component configuration is not satisfied, unless all its references are satisfied. A reference specifies target services by specifying their interface and an optional target property.

A reference element has the attributes defined in the following table.

Table 112.8 Reference Element and Annotations

Attribute	Annotation	Description
name	name	The name of the reference. This name is local to the component and can be used to locate a bound service of this reference with one of the locateService methods of ComponentContext. Each reference element within the component must have a unique name. This name attribute is optional. The default value of this attribute is the value of the interface attribute of this element. If multiple reference elements in the component use the same interface name, then using the default value for this attribute will result in duplicate reference names. In this case, this attribute must be specified with a unique name for the reference to avoid an error. The Reference annotation will use the name of the annotated method, field, or parameter as the default reference name. If the method name begins with bind, set or add, that prefix is removed.

Attribute	Annotation	Description
interface	service	Fully qualified name of the class that is used by the component to access the service. The service provided to the component must be type compatible with this class. That is, the component must be able to cast the service object to this class. A service must be registered under this name to be considered for the set of target services. The Reference annotation will use the type of the first parameter of the annotated method or the type of the annotated parameter or field to determine the service value. The special org.osgi.service.component.AnyService service name can be used to indicate that the service type name is not used to select target services. Only the target property is used to select target services. When using this special service name, the reference field or parameter must be of type java.lang.Object so that any service object can be provided. See *Any Service Type* on page 262.
cardinality	cardinality MANDATORY OPTIONAL MULTIPLE AT_LEAST_ONE	Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services. See *Reference Cardinality* on page 257.
policy	policy STATIC DYNAMIC	The policy declares the assumption of the component about dynamicity. See *Reference Policy* on page 259.
policy-option	policyOption RELUCTANT GREEDY	Defines the policy when a better service becomes available. See *Reference Policy* on page 259.
target	target	An optional OSGi Framework filter expression that further constrains the set of target services. The default is no filter, limiting the set of matched services to all service registered under the given reference interface. The value of this attribute is used for the value of the target property of the reference. See *Target Property* on page 287.
scope	scope BUNDLE PROTOTYPE PROTOTYPE_ REQUIRED	The reference scope for this reference. See *Reference Scope* on page 258.
bind	Reference bind	The name of a method in the component implementation class that is used to notify that a service is bound to the component configuration. For static references, this method is only called before the activate method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 251. The Reference annotation will use the name of the method it is applied to as the bind method name.
updated	updated	The name of a method in the component implementation class that is used to notify that a bound service has modified its properties.

Attribute	Annotation	Description
unbind	unbind	Same as bind, but is used to notify the component configuration that the service is unbound. For static references, the method is only called after the deactivate method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 251.
field	Reference field	The name of a field in the component implementation class that is used to hold service(s) that are bound to the component configuration. For static references, this field is set once after the constructor has been called and before calling the activate method. For dynamic references, this field can modified while the component configuration is active. See *Accessing Services* on page 251. The Reference annotation will use the name of the field it is applied to as the field name.
field-option	fieldOption REPLACE UPDATE	Defines how the field value must be managed. This is ignored if the field attribute is not set. See *Reference Field Option* on page 260.
field-collection-type	collectionType SERVICE REFERENCE SERVICEOBJECTS PROPERTIES TUPLE	Defines the types of elements in the collection or Optional referenced by the field value or constructor parameter. This is ignored if the field attribute or parameter attribute is not set. It is also ignored when the field attribute or parameter attribute is set, the cardinality is unary, and the field or constructor parameter type is not Optional. See *Field Injection* on page 254 for more information. The Reference annotation can generally infer the value of the collection elements from the generic type information of the annotated field or constructor parameter but it can be explicitly defined if needed.
parameter	Reference parameter	The zero-based parameter number of a parameter in the constructor of the component that is used to receive service(s) that are bound to the component configuration. If this attribute is set and the policy attribute is set to DYNAMIC, this attribute must be ignored and SCR must log an error message with the Log Service, if present. See *Accessing Services* on page 251. The Reference annotation will use the zero-based parameter number of the parameter it is applied to as the parameter number.

In the generated component description, the reference elements must be ordered in ascending lexicographical order, using String.compareTo, of the names of the references.

The following code demonstrates the use of the Reference annotation for method injection.

```
@Component
public class FooImpl implements Foo {
  @Reference(
    policy = DYNAMIC,
    policyOption = GREEDY,
    cardinality = MANDATORY )
  void setLog( LoggerFactory lf ) { ... }
  void unsetLog( LoggerFactory lf ) { ... }
  void updatedLog( Map<String,?> ref ) { ... }

  @Activate
  void open() { ... }
  @Deactivate
  void close() { ... }
}
```

The following code demonstrates the use of the Reference annotation for field injection.

```
@Component
public class FooImpl implements Foo {
  @Reference
  volatile LoggerFactory lf;

  @Activate
  void open() { lf.getLogger(FooImpl.class).info("activated"); }
  @Deactivate
  void close() { lf.getLogger(FooImpl.class).info("deactivated"); }
}
```

The following code demonstrates the use of the Reference annotation for constructor injection.

```
@Component
public class FooImpl implements Foo {
  private final Logger logger;

  @Activate
  public FooImpl( @Reference LoggerFactory lf ) {
    logger = lf.getLogger(FooImpl.class);
  }

  @Activate
  void open() { logger.info("activated"); }
  @Deactivate
  void close() { logger.info("deactivated"); }
}
```

For a reference to be used with the lookup strategy, there are no bind methods or fields to annotate with the Reference annotation. Instead Reference annotations can be specified in the reference element of the Component annotation. When used in this way, the name and service elements must be specified since there is no annotated member from which the name or service can be determined. The following code demonstrates the use of the Reference annotation for the lookup strategy.

```
@Component( reference =
  @Reference( name = "log", service = LoggerFactory.class )
)
public class FooImpl implements Foo {
  @Activate
  void open( ComponentContext context ) {
    LoggerFactory lf = context.locateService( "log" );
    ...
  }
  @Deactivate
  void close() { ... }
}
```

112.4.9 Factory Property and Factory Properties Elements

If the component is a factory component, see *Factory Component* on page 250, the component description can define a number of factory properties. These can be defined inline or from a resource in the bundle. The factory-property and factory-properties elements can occur multiple times and they can be interleaved. This interleaving is relevant because the factory properties are processed from top to bottom. Later factory properties override earlier factory properties that have the same name.

The factory-property element has the attributes and annotations defined in the following table.

Table 112.9 Factory Property Element and Annotations

Attribute	Annotation	Description
name	Component factoryProperty	The name of the factory property.
value		The value of the factory property. This value is parsed according to the property type. If the value attribute is specified, the body of the element is ignored. If the type of the factory property is not String, parsing of the value is done by the static valueOf(String) method in the given type. For Character types, the conversion must be handled by Integer.valueOf method, a Character is always represented by its Unicode value.
type		The type of the factory property. Defines how to interpret the value. The type must be one of the following Java types: • String (default) • Long • Double • Float • Integer • Byte • Character • Boolean • Short
<body>		If the value attribute is not specified, the body of the factory-property element must contain one or more values. The value of the factory property is then an array of the specified type. The result will be translated to an array of primitives or Strings. For example, if the type attribute specifies Integer, then the resulting array must be int[]. Values must be placed one per line and blank lines are ignored. Parsing of each value is done by the parse methods in the class identified by the type, after trimming the line of any beginning and ending white space. Each String value is also trimmed of beginning and ending white space.

A factory property can also be set with the factoryProperty annotation element of Component. This element is an array of strings that must follow the following syntax:

```
factory-property ::= name ( ':' type )? '=' value
```

In this case name, type, and value parts map to the attributes of the factory-property element. If multiple values must be specified then the same name can be repeated multiple times.

The factory-properties element references an entry in the bundle whose contents conform to a standard [3] *Java Properties File*.

At runtime, SCR reads the entry to obtain the factory properties and their values. The factory-properties element attributes are defined in the following table.

Table 112.10 Factory Properties Element and Annotations

Attribute	Annotation	Description
entry	Component factoryProperties	The entry path relative to the root of the bundle

For example, to include properties that are stored in the OSGI-INF directory, the following definition could be used:

```
<factory-properties entry="OSGI-INF/factory.properties"/>
```

The factoryProperties annotation element of Component can be used to provide the same information. This element consists of an array of strings where each string defines an entry. The order within the array is the order that must be used for the XML. However, the annotations do not support interleaving of the generated factory-property and factory-properties elements.

For example:

```
@Component(factoryProperties="OSGI-INF/factory.properties")
```

When using annotation elements to specify factory properties, a tool processing the Component annotations must write the defined factory properties into the generated component description in the following order.

1. factoryProperty element of the Component annotation.
2. factoryProperties element of the Component annotation.

112.5 Component Life Cycle

112.5.1 Enabled

A component must first be *enabled* before it can be used. A component cannot be enabled unless the component's bundle is started. See *Starting Bundles* in *OSGi Core Release 8*. All components in a bundle become disabled when the bundle is stopped. So the life cycle of a component is contained within the life cycle of its bundle.

Every component can be enabled or disabled. The initial enabled state of a component is specified in the component description via the enabled attribute of the component element. See *Component Element* on page 266. Component configurations can be created, satisfied and activated only when the component is enabled.

The enabled state of a component can be controlled with the Component Context enableComponent(String) and disableComponent(String) methods. The purpose of later enabling a component is to be able to decide programmatically when a component can become enabled. For example, an immediate component can perform some initialization work before other components in the bundle are enabled. The component descriptions of all other components in the bundle can be disabled by having enabled set to false in their component descriptions. After any necessary initialization work is complete, the immediate component can call enableComponent to enable the remaining components.

The enableComponent and disableComponent methods must return after changing the enabled state of the named component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to the method call. Therefore a component can disable itself.

All components in a bundle can be enabled by passing a null as the argument to enableComponent.

112.5.2 Satisfied

Component configurations can only be activated when the component configuration is *satisfied*. A component configuration becomes satisfied when the following conditions are all satisfied:

- The component is *enabled*.

- If the component description specifies `configuration-policy=required`, then a `Configuration` object for the component is present in the Configuration Admin service.
- Using the component properties of the component configuration, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or the number of target services is equal to or more than the minimum cardinality of the reference.

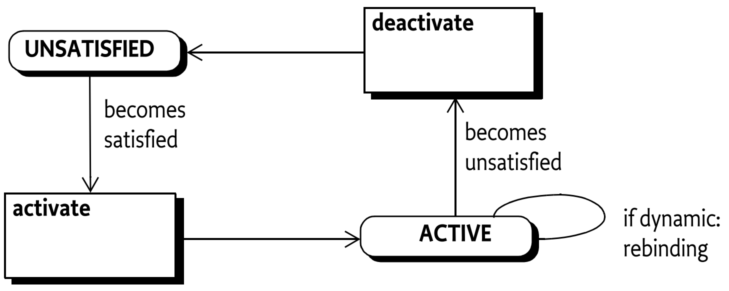
Once any of the listed conditions are no longer true, the component configuration becomes *unsatisfied*. An activated component configuration that becomes unsatisfied must be deactivated.

112.5.3 Immediate Component

A component is an immediate component when it must be activated as soon as its dependencies are satisfied. Once the component configuration becomes unsatisfied, the component configuration must be deactivated. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 286.

The state diagram is shown in Figure 112.2.

Figure 112.2 *Immediate Component Configuration*



112.5.4 Delayed Component

A key attribute of a delayed component is the delaying of class loading and object creation. Therefore, the activation of a delayed component configuration does not occur until there is an actual request for a service object. A component is a delayed component when it specifies a service but it is not a factory component and does not have the `immediate` attribute of the `component` element set to `true`.

SCR must register a service after the component configuration becomes satisfied. The registration of this service must look to observers of the service registry as if the component's bundle actually registered this service. This makes it possible to register services without creating a class loader for the bundle and loading classes, thereby allowing reduction in initialization time and a delay in memory footprint.

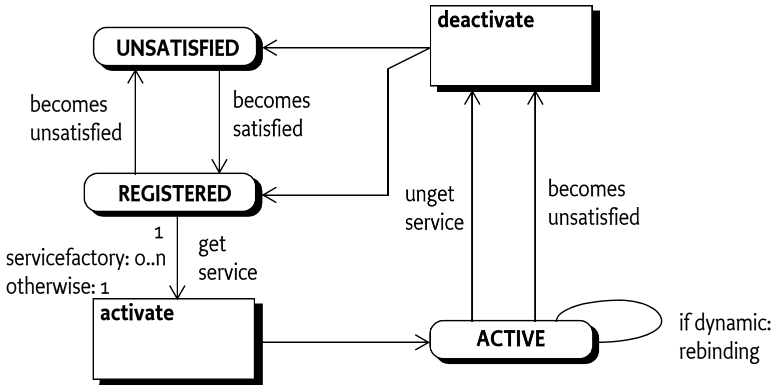
When SCR registers the service on behalf of a component configuration, it must avoid causing a class load to occur from the component's bundle. SCR can ensure this by registering a `ServiceFactory` object with the Framework for that service. By registering a `ServiceFactory` object, the actual service object is not needed until the `ServiceFactory` is called to provide the service object. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 286.

The activation of a component configuration must be delayed until its service is requested. When the service is requested, if the service has the `scope` attribute set to `bundle`, SCR must create and activate a unique component configuration for each bundle requesting the service. If the service has the `scope` attribute set to `prototype`, SCR must create and activate a unique component configuration for each distinct request for the service. Otherwise, if the service has the `scope` attribute set to singleton, SCR must activate a single component configuration which is used by all requests for the service. A component instance can determine the bundle it was activated for by calling the getUsingBundle() method on the Component Context.

The activation of delayed components is depicted in a state diagram in Figure 112.3. Notice that multiple component configurations can be created from the REGISTERED state if a delayed component specifies a service scope set to a value other than singleton.

If the service has the scope attribute set to prototype, SCR must deactivate a component configuration when it stops being used as a service object since the component configuration must not be reused as a service object. If the service has the scope attribute set to singleton or bundle, SCR must deactivate a component configuration when it stops being used as a service object after a delay since the component configuration may be reused as a service object in the near future. This allows SCR implementations to reclaim component configurations not in use while attempting to avoid deactivating a component configuration only to have to quickly activate a new component configuration for a new service request. The delay amount is implementation specific and may be zero.

Figure 112.3 Delayed Component Configuration



112.5.5 Factory Component

SCR must register a Component Factory service as soon as the *component factory* becomes satisfied. The component factory is satisfied when the following conditions are all satisfied:

- The component is enabled.
- Using the component properties specified by the component description, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or there is at least one target service for the reference.

The component factory, however, does not use any of the target services and does not bind to them.

Once any of the listed conditions are no longer true, the component factory becomes unsatisfied and the Component Factory service must be unregistered. Any component configurations activated via the component factory are unaffected by the unregistration of the Component Factory service, but may themselves become unsatisfied for the same reason.

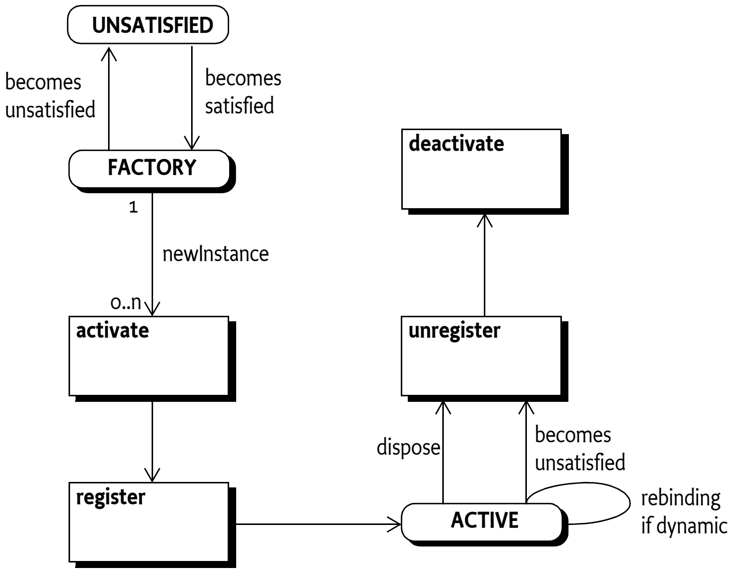
The Component Factory service must be registered under the name org.osgi.service.component.ComponentFactory with the following service properties:

- component.name - The name of the component.
- component.factory - The value of the factory attribute.

The service properties of the Component Factory service must not include the component properties.

New component configurations are created and activated when the newInstance method of the Component Factory service is called. If the component description specifies a service, the component configuration is registered as a service under the provided interfaces. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 286. The service registration must take place before the component configuration is activated. Service unregistration must take place before the component configuration is deactivated.

Figure 112.4 *Factory Component*



A Component Factory service has a single method: newInstance(Dictionary). This method must create, satisfy and activate a new component configuration and register its component instance as a service if the component description specifies a service. It must then return a ComponentInstance object. This ComponentInstance object can be used to get the component instance with the getInstance() method.

SCR must attempt to satisfy the component configuration created by newInstance before activating it. If SCR is unable to satisfy the component configuration given the component properties and the Dictionary argument to newInstance, the newInstance method must throw a ComponentException.

The client of the Component Factory service can also deactivate a component configuration with the dispose() method on the ComponentInstance object. If the component configuration is already deactivated, or is being deactivated, then this method is ignored. Also, if the component configuration becomes unsatisfied for any reason, it must be deactivated by SCR.

Once a component configuration created by the Component Factory has been deactivated, that component configuration will not be reactivated or used again.

112.5.6 Activation

Activating a component configuration consists of the following steps:

1. Load the component implementation class.
2. Compute the bound services. See *Bound Services* on page 280.
3. Create the component context. See *Component Context* on page 280.
4. Construct the component instance. See *Constructor Injection* on page 256.
5. Set the activation fields, if any. See *Activation Objects* on page 280.
6. Bind the bound services. See *Binding Services* on page 280.

7. Call the `activate` method, if any. See *Activate Method* on page 281. Calling the `activate` method signals the completion of activating the component instance.

Component instances must never be reused. Each time a component configuration is activated, SCR must create a new component instance to use with the activated component configuration. A component instance must complete activation before it can be deactivated. Once the component configuration is deactivated or fails to activate due to an exception, SCR must unbind all the component's bound services and discard all references to the component instance associated with the activation.

112.5.7 Bound Services

When a component configuration's reference is satisfied, there is a set of zero or more target services for that reference. When the component configuration is activated, a subset of the target services for each reference are bound to the component configuration. The subset is chosen by the cardinality of the reference. See *Reference Cardinality* on page 257.

Obtaining the service object for a bound service may result in activating a component configuration of the bound service which could result in an exception. If the loss of the bound service due to the exception causes the reference's cardinality constraint to be violated, then activation of this component configuration will fail. Otherwise the bound service which failed to activate will be considered unbound.

112.5.8 Component Context

The Component Context can be made available to a component instance during activation, modification, and deactivation. It provides the interface to the execution context of the component, much like the Bundle Context provides a bundle the interface to the Framework. A Component Context should therefore be regarded as a capability and not shared with other components or bundles.

Each distinct component instance receives a unique Component Context. Component Contexts are not reused and must be discarded when the component configuration is deactivated.

112.5.9 Activation Objects

A component can have an `activate` method, activation fields, and also receive activation objects via its constructor.

The following *activation object* types are supported:

- `ComponentContext` - The Component Context for the component configuration.
- `BundleContext` - The Bundle Context of the component's bundle.
- `Map` - An unmodifiable Map containing the component properties.
- A component property type - An instance of the component property type which allows type safe access to component properties defined by the component property type. See *Component Property Types* on page 292.

For activation fields, only instance fields of the activation object types above are supported. If an activation field is declared with the `static` modifier or has a type other than one of the above, SCR must log an error message with the Log Service, if present, and the field must not be modified. SCR must locate a suitable field as specified in *Locating Component Methods and Fields* on page 298. If no suitable field is located for an activation field name, SCR must log an error message with the Log Service, if present.

112.5.10 Binding Services

When binding services, the references are processed in the order in which they are specified in the component description. That is, target services from the first specified reference are bound before services from the next specified reference.

If the reference uses field injection, the field must be set. Then, if the reference uses method injection, the bind method must be called for each bound service of that reference. If a bind method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, but the activation of the component configuration does not fail.

112.5.11 Activate Method

A component can have an `activate` method. The name of the `activate` method can be specified by the `activate` attribute. If the `activate` attribute is not specified, the default method name of `activate` is used. See *Component Element* on page 266.

The activate method can take zero or more parameters. Each parameter must be assignable from one of the activation object types. A suitable method is selected using the following priority:

1. The method takes a single parameter and the type of the parameter is `org.osgi.service.component.ComponentContext`.
2. The method takes a single parameter and the type of the parameter is `org.osgi.framework.BundleContext`.
3. The method takes a single parameter and the type of the parameter is a component property type.
4. The method takes a single parameter and the type of the parameter is `java.util.Map`.
5. The method takes two or more parameters and the type of each parameter must be one of the activation object types. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
6. The method takes zero parameters.

When searching for the activate method to call, SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 298. If the `activate` attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration is not activated.

If an activate method is located, SCR must call this method to complete the activation of the component configuration. If the activate method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the component configuration is not activated.

112.5.12 Bound Service Replacement

If an active component configuration has a dynamic reference with unary cardinality and the bound service is modified or unregistered and ceases to be a target service, or the `policy-option` is `greedy` and a better target service becomes available then SCR must attempt to replace the bound service with a new bound service.

If the reference uses field injection, the field must be set for the replacement bound service. Then, if the reference uses method injection, SCR must first bind the new bound service and then unbind the outgoing service. This reversed order allows the component to not have to handle the inevitable gap between the unbind and bind methods. However, this means that in the unbind method care must be taken to not overwrite the newly bound service. For example, the following code handles the associated concurrency issues and simplify handling the reverse order.

```
final AtomicReference<LogService> log = new AtomicReference<LogService>();

void setLogService( LogService log ) {
    this.log.set(log);
}
void unsetLogService( LogService log ) {
    this.log.compareAndSet(log, null);
}
```

If the dynamic reference falls below the minimum cardinality, the component configuration must be deactivated because the cardinality constraints will be violated.

If a component configuration has a static reference and a bound service is modified or unregistered and ceases to be a target service, or the `policy-option` is `greedy` and a better target service becomes available then SCR must deactivate the component configuration. Afterwards, SCR must attempt to activate the component configuration again if another target service can be used as a replacement for the outgoing service.

112.5.13 Updated

If an active component is bound to a service that modifies its service properties then the component can be updated. If the reference uses field injection and the field holds the service properties, the field must be set for the updated bound service. Then, if the reference uses method injection and specifies an updated method, the updated method must be called.

112.5.14 Modification

Modifying a component configuration can occur if the component description specifies the `modified` attribute and the component properties of the component configuration use a `Configuration` object from the Configuration Admin service and that `Configuration` object is modified without causing the component configuration to become unsatisfied. If this occurs, the component instance will be notified of the change in the component properties.

If the `modified` attribute is not specified, then the component configuration will become unsatisfied if its component properties use a `Configuration` object and that Configuration object is modified in any way.

Modifying a component configuration consists of the following steps:

1. Update the component context for the component configuration with the modified configuration properties.
2. Call the modified method. See *Modified Method* on page 282.
3. Modify the bound services for the dynamic references if the set of target services changed due to changes in the target properties. See *Bound Service Replacement* on page 281.
4. If the component configuration is registered as a service, modify the service properties.

A component instance must complete activation, or a previous modification, before it can be modified.

See *Configuration Changes* on page 288 for more information.

112.5.15 Modified Method

The name of the modified method is specified by the `modified` attribute. See *Component Element* on page 266.

The modified method can take zero or more parameters. Each parameter must be assignable from one of the activation object types. A suitable method is selected using the following priority:

1. The method takes a single parameter and the type of the parameter is `org.osgi.service.component.ComponentContext`.
2. The method takes a single parameter and the type of the parameter is `org.osgi.framework.BundleContext`.
3. The method takes a single parameter and the type of the parameter is a component property type.
4. The method takes a single parameter and the type of the parameter is `java.util.Map`.

5. The method takes two or more parameters and the type of each parameter must be one of the activation object types. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
6. The method takes zero parameters.

SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 298. If the `modified` attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration becomes unsatisfied and is deactivated as if the `modified` attribute was not specified.

If a modified method is located, SCR must call this method to notify the component configuration of changes to the component properties. If the modified method throws an exception, SCR must log an error message containing the exception with the Log Service, if present and continue processing the modification.

112.5.16 Deactivation

Deactivating a component configuration consists of the following steps:

1. Call the deactivate method, if present. See *Deactivate Method* on page 283.
2. Unbind any bound services. See *Unbinding* on page 284.
3. Release all references to the component instance and component context.

A component instance must complete activation or modification before it can be deactivated. A component configuration can be deactivated for a variety of reasons. The deactivation reason can be received by the deactivate method. The following reason values are defined:

- `DEACTIVATION_REASON_UNSPECIFIED` - Unspecified.
- `DEACTIVATION_REASON_DISABLED` - The component was disabled.
- `DEACTIVATION_REASON_REFERENCE` - A reference became unsatisfied.
- `DEACTIVATION_REASON_CONFIGURATION_MODIFIED` - A configuration was changed.
- `DEACTIVATION_REASON_CONFIGURATION_DELETED` - A configuration was deleted.
- `DEACTIVATION_REASON_DISPOSED` - The component was disposed.
- `DEACTIVATION_REASON_BUNDLE_STOPPED` - The bundle was stopped.

Once the component configuration is deactivated, SCR must discard all references to the component instance and component context associated with the activation.

112.5.17 Deactivate Method

A component instance can have a deactivate method. The name of the deactivate method can be specified by the `deactivate` attribute. See *Component Element* on page 266. If the `deactivate` attribute is not specified, the default method name of `deactivate` is used. Activation fields must not be modified during deactivation.

The deactivate method can take zero or more parameters. Each parameter must be assignable from one of the following types:

- One of the activation object types.
- `int` or `Integer` - The reason the component configuration is being deactivated. See *Deactivation* on page 283.

A suitable method is selected using the following priority:

1. The method takes a single parameter and the type of the parameter is `org.osgi.service.component.ComponentContext`.
2. The method takes a single parameter and the type of the parameter is `org.osgi.framework.BundleContext`.

3. The method takes a single parameter and the type of the parameter is a component property type.
4. The method takes a single parameter and the type of the parameter is `java.util.Map`.
5. The method takes a single parameter and the type of the parameter is `int`.
6. The method takes a single parameter and the type of the parameter is `java.lang.Integer`.
7. The method takes two or more parameters and the type of each parameter must be one of the activation object types, `int` or `java.lang.Integer`. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
8. The method takes zero parameters.

When searching for the deactivate method to call, SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 298. If the `deactivate` attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the deactivation of the component configuration will continue.

If a deactivate method is located, SCR must call this method to commence the deactivation of the component configuration. If the deactivate method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the deactivation of the component configuration will continue.

112.5.18 Unbinding

When a component configuration is deactivated, the bound services are unbound from the component configuration.

When unbinding services, the references are processed in the reverse order in which they are specified in the component description. That is, target services from the last specified reference are unbound before services from the previous specified reference.

If the reference uses method injection, the unbind method must be called for each bound service of that reference. If an unbind method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the deactivation of the component configuration will continue. Then, if the reference uses field injection, the field must be set to `null`.

112.5.19 Life Cycle Example

A component could declare a dependency on the Http Service to register some resources.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.binding"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
    <implementation class="com.acme.impl.Binding"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LogService"
        cardinality="1..1"
        policy="static"
    />
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
        cardinality="0..1"
        policy="dynamic"
        bind="setHttp"
        unbind="unsetHttp"
    />
</scr:component>
```

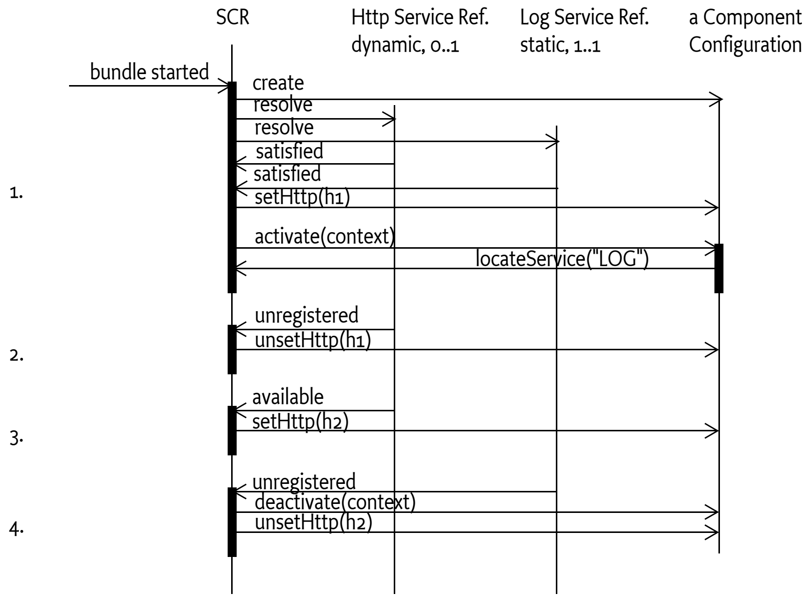
The component implementation code looks like:

```
public class Binding {
    LogService  log;
    HttpService http;

    private void setHttp(HttpService h) {
        http = h;
        // register servlet
    }
    private void unsetHttp(HttpService h){
        if (http == h)
            http = null;
        // unregister servlet
    }
    private void activate(ComponentContext context ) {
        log = (LogService) context.locateService("LOG");
    }
    private void deactivate(ComponentContext context ) {...}
}
```

This example is depicted in a sequence diagram in Figure 112.5 with the following scenario:

1. A bundle with the `example.Binding` component is started. At that time there is a Log Service l1 and a Http Service h1 registered.
2. The Http Service h1 is unregistered
3. A new Http Service h2 is registered
4. The Log Service h1 is unregistered.

Figure 112.5 *Sequence Diagram for binding*



112.6 Component Properties

Each component configuration is associated with a set of component properties. The component properties are specified in the following *configuration sources* (in order of precedence):

1. Properties specified in the argument of the ComponentFactory.newInstance method. This is only applicable for factory components.
2. Properties retrieved from the OSGi Configuration Admin service in Configuration objects whose PID matches a *configuration PID*. The configuration PIDs are specified by the configuration-pid attribute of the component element. See *Component Element* on page 266. If no configuration-pid attribute is specified, the component name is used as the default configuration PID. If multiple configuration PIDs are specified, the order of precedence follows the order the configuration PIDs are specified in the component description. That is, the precedence for the configuration for an earlier specified configuration PID is lower than the precedence for the configurations for a later specified configuration PID.
3. Properties specified in the component description via property and properties elements. Properties specified later in the component description override properties that have the same name specified earlier. See *Property and Properties Elements* on page 268.
4. Target properties specified in the component description via the target attribute of reference elements. See *Target Property* on page 287. The value of the target attribute is used for the value of a target property.

The precedence behavior allows certain default values to be specified in the component description while allowing properties to be replaced and extended by:

- A configuration in Configuration Admin
- The argument to the ComponentFactory.newInstance method

Normally, a property value from a higher precedence configuration source replace a property value from a lower precedence configuration source. However, the service.pid property values receive different treatment. For the service.pid property, if the property appears multiple times in the configuration sources, SCR must aggregate all the values found into a Collection<String> having an iteration order such that the first item in the iteration is the property value from the lowest precedence configuration source and the last item in the iteration is the property value from the highest precedence configuration source. If the component description specifies multiple configuration PIDs, then the order of the service.pid property values from the corresponding configurations matches the order the configuration PIDs are specified in the component description. The values of the service.pid component property are the values as they come from the configuration sources which, for Configuration objects, may be more detailed than the configuration PIDs specified in the component description.

SCR always adds the following component properties, which cannot be overridden:

- component.name - The component name.
- component.id - A unique value (Long) that is larger than all previously assigned values. These values are not persistent across restarts of SCR.

112.6.1 Service Properties

When SCR registers a service on behalf of a component configuration, SCR must follow the recommendations in *Property Propagation* on page 73 and must not propagate private configuration properties. That is, the service properties of the registered service must be all the component properties of the component configuration whose property names do not start with full stop ('.' \u002E).

Component properties whose names start with full stop are available to the component instance but are not available as service properties of the registered service.

112.6.2 Reference Properties

This specification defines some component properties which are associated with specific component references. These are called *reference properties*. The name of a reference property for a reference is the name of the reference appended with a full stop ('.' \u002E) and a suffix unique to the reference property. Reference properties can be set wherever component properties can be set.

All component property names starting with a reference name followed by a full stop ('.' \u002E) are reserved for use by this specification.

Following are the reference properties defined by this specification.

112.6.2.1 Target Property

The *target property* is a reference property which aids in the selection of target services for the reference. See *Selecting Target Services* on page 262. The name of a target property is the name of a reference appended with `.target`. For example, the target property for a reference with the name `http` would have the name `http.target`. The value of a target property is a filter string used to select targets services for the reference.

The target property for a reference can also be set by the `target` attribute of the `reference` element. See *Reference Element* on page 271.

112.6.2.2 Minimum Cardinality Property

The initial minimum cardinality of a reference is specified by the optionality: the first part of the cardinality. It is either `0` or `1`. The minimum cardinality of a reference cannot exceed the multiplicity: the second part of the cardinality. See *Reference Cardinality* on page 257 for more information on the cardinality of a reference.

The *minimum cardinality property* is a reference property which can be used to raise the minimum cardinality of a reference from its initial value. That is, a `0..1` cardinality can be raised to a `1..1` cardinality by setting the reference's minimum cardinality property to `1`, and a `0..n` or `1..n` cardinality can be raised to a `m..n` cardinality by setting the reference's minimum cardinality property to `m` such that `m` is a positive integer. The minimum cardinality of a reference cannot be lowered. That is, a `1..1` or `1..n` cardinality cannot be lowered to a `0..1` or `0..n` cardinality because the component was coded to expect at least one bound service.

The name of a minimum cardinality property is the name of a reference appended with `.cardinality.minimum`. For example, the minimum cardinality property for a reference with the name `http` would have the name `http.cardinality.minimum`. The value of a minimum cardinality property must be a positive integer or a value that can be coerced into a positive integer. See *Coercing Component Property Values* on page 295 for information on coercing property values. If the numerical value of the minimum cardinality property is not valid for the reference's cardinality or the minimum cardinality property value cannot be coerced into a numerical value, then the minimum cardinality property must be ignored.

SCR must support the minimum cardinality property for all components even those with component descriptions in older namespaces.

112.7 Deployment

A component description contains default information to select target services for each reference. However, when a component is deployed, it is often necessary to influence the target service selection in a way that suits the needs of the deployer. Therefore, SCR uses `Configuration` objects from Configuration Admin to replace and extend the component properties for a component configuration. That is, through Configuration Admin, a deployer can configure component properties.

A component's configuration PIDs are used as keys for obtaining additional component properties from Configuration Admin. When *matching* a configuration PID to a `Configuration` object, SCR must use the `Configuration` object with the best matching PID for the component's bundle. See *Targeted PIDs* on page 69 for more information on targeted PIDs and *Extenders and Targeted PIDs* on page 70 for more information on selecting the `Configuration` object with the best matching PID.

The following situations can arise when looking for Configuration objects:

- *No Configuration* - If the component's configuration-policy is set to ignore or there are no Configurations with a PID or factory PID matching any of the configuration PIDs, then component configurations will not obtain component properties from Configuration Admin. Only component properties specified in the component description or via the ComponentFactory.newInstance method will be used.
- *Not Satisfied* - If the component's configuration-policy is set to require and, for each configuration PID, there is no Configuration with a matching PID or factory PID, then the component configuration is not satisfied and will not be activated.
- *Single Configuration* - If none of the configuration PIDs matches a factory PID, then component configurations will obtain additional component properties from Configuration Admin.
- *Factory Configuration* - If one of the configuration PIDs matches a factory PID, with zero or more Configurations, then for each Configuration of the factory PID, a component configuration must be created that will obtain additional component properties from Configuration Admin.

  It is a configuration error if more than one of the configuration PIDs match a factory PID and SCR must log an error message with the Log Service, if present. If the configuration-policy is set to optional, the component configuration must be satisfied without the configurations PIDs which match a factory PID. If the configuration-policy is set to require, the component configuration is not satisfied and will not be activated.

  A factory configuration must not be used if the component is a factory component. This is because SCR is not free to create component configurations as necessary to support multiple Configurations. When SCR detects this condition, it must log an error message with the Log Service, if present, and ignore the component description.

SCR must obtain the Configuration objects from the Configuration Admin service using the Bundle Context of the bundle containing the component. SCR must only use Configuration objects for which the bundle containing the component has visibility. See *Location Binding* on page 71.

To ensure Configuration Plugins can participate in the configuration process, SCR must use the Configuration.getProcessedProperties method when obtaining the configuration data from a Configuration object. To use the getProcessedProperties method, SCR must supply a Service Reference for a ManagedService or ManagedServiceFactory service. The ManagedService or ManageServiceFactory service must be registered using the Bundle Context of the bundle containing the component. If SCR registers one of these services for the purpose of using the service's Service Reference for the call to getProcessedProperties, SCR should register the service without a service.pid service property so that the service itself is not called by Configuration Admin.

For example, there is a component named com.acme.client with a reference named HTTP that requires an Http Service which must be bound to a component com.acme.httpserver which provides an Http Service. A deployer can establish the following configuration:

```
[PID=com.acme.client, factoryPID=null]
HTTP.target = (component.name=com.acme.httpserver)
```

112.7.1 Configuration Changes

SCR must track changes in the Configuration objects matching the configuration PIDs of a component description. Changes include the creating, updating and deleting of Configuration objects matching the configuration PIDs. The actions SCR must take when a configuration change for a component configuration occurs are based upon how the configuration-policy and modified attributes are specified in the component description, whether a component configuration becomes satisfied, remains satisfied or becomes unsatisfied and the type and number of matching Configuration objects.

With targeted PIDs, multiple Configuration objects can exist which can match a configuration PID. Creation of a Configuration object with a better matching PID than a Configuration object currently being used by a component configuration results in a configuration change for the component configuration with the new Configuration object replacing the currently used Configuration object. Deletion of a Configuration object currently being used by a component configuration when there is another Configuration object matching the configuration PID also results in a configuration change for the component configuration with the Configuration object having the best matching PID replacing the currently used, and now deleted, Configuration object.

112.7.1.1 Ignore Configuration Policy

For configuration-policy of ignore, component configurations are unaffected by configuration changes since the component properties do not include properties from Configuration objects.

112.7.1.2 Require Configuration Policy

For configuration-policy of require, component configurations require a Configuration object for each specified configuration PID. With a factory configuration, there can be zero or more matching Configuration objects which will result in a component configuration for each Configuration object of the factory configuration. With a factory component, multiple component configurations can be created all using the matching Configuration objects.

A configuration change can cause a component configuration to become unsatisfied if any of the following occur:

- Each configuration PID of the component description does not have a matching Configuration object.
- A target property change results in a bound service of a static reference ceasing to be a target service.
- A target property change results in unbound target services for a static reference with the greedy policy option.
- A target property change or minimum cardinality property change results in a reference falling below the minimum cardinality.
- The component description does not specify the modified attribute.

112.7.1.3 Optional Configuration Policy

For configuration-policy of optional, component configurations do not require Configuration objects. Since matching Configuration objects are optional, component configurations can be satisfied with zero or more matched configuration PIDs. If a Configuration object is then created which matches a configuration PID, this is a configuration change for the component configurations that are not using the created Configuration object. If a Configuration object is deleted which matches a configuration PID, this is a configuration change for the component configurations using the deleted Configuration object.

Furthermore, with a factory configuration matching a configuration PID, the factory configuration can provide zero or more Configuration objects which will result in a component configuration for each Configuration object or a single component configuration when zero matching Configuration objects are provided. With a factory component, multiple component configurations can be created all using the Configuration objects matching the configuration PIDs.

A configuration change can cause a component configuration to become unsatisfied if any of the following occur:

- A target property change results in a bound service of a static reference ceasing to be a target service.
- A target property change results in unbound target services for a static reference with the greedy policy option.

- A target property change or minimum cardinality property change results in a reference falling below the minimum cardinality.
- The component description does not specify the modified attribute.

112.7.1.4 Configuration Change Actions

If a component configuration becomes unsatisfied:

- SCR must deactivate the component configuration.
- If the component configuration was not created from a factory component, SCR must attempt to satisfy the component configuration with the current configuration state.

If a component configuration remains satisfied:

- If the component configuration has been activated, the modified method is called to provide the updated component properties. See *Modification* on page 282 for more information.
- If the component configuration is registered as a service, SCR must modify the service properties.

112.7.1.5 Coordinator Support

The *Coordinator Service Specification* on page 613 defines a mechanism for multiple parties to collaborate on a common task without *a priori* knowledge of who will collaborate in that task. Like *Configuration Admin Service Specification* on page 65, SCR must participate in such scenarios to coordinate with provisioning or configuration tasks.

If configurations changes occur and an implicit coordination exists, SCR must delay taking action on the configuration changes until the coordination terminates, regardless of whether the coordination fails or terminates regularly.

112.8 Annotations

A number of CLASS retention annotations have been provided to allow tools to construct the component description XML from the Java class files. The Component Annotations are intended to be used during build time to generate the component description XML.

Component Property Types, which are user defined annotations, can be used to describe component properties in the component description XML and to access those component properties at runtime in a type safe manner.

112.8.1 Component Annotations

The Component Annotations provide a convenient way to create the component description XML during build time. Since annotations are placed in the source file and can use types, fields, and methods, they can significantly simplify the use of Declarative Services.

The Component Annotations are build time annotations because one of the key aspects of Declarative Services is its laziness. SCR can easily read the component description XML from the bundle, preprocess it, and cache the results between framework invocations. This way it is unnecessary to load a class from the bundle when the bundle is started and/or scan the classes for annotations. Component Annotations are not recognized by SCR at runtime.

The Component Annotations are not inherited, they can only be used on a given class, annotations on its super class hierarchy or interfaces are not taken into account.

The primary annotation is the Component annotation. It indicates that a class is a component. Its defaults create the easiest to use component:

- Its name is the class name
- It registers all of the class's directly implemented interfaces as services

- The instance will be shared by all bundles
- It is enabled
- It is immediate if it has no services, otherwise it is delayed
- It has an optional configuration policy
- The configuration PID is the class name

For example, the following class registers a Speech service that can run on a Macintosh:

```
public interface Speech {
  void say(String what) throws Exception;
}

@Component
public class MacSpeech implements Speech {
    ScriptEngine engine =
        new ScriptEngineManager().getEngineByName("AppleScript");

    public void say(String message) throws Exception {
        engine.eval("say \"" + message.replace('"','\'' + "\"");
    }
}
```

The previous example would be processed at build time into a component description similar to the following XML:

```
<scr:component name="com.example.MacSpeech"
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0">
  <implementation class="com.acme.impl.MacSpeech"/>
  <service>
    <provide interface="com.acme.service.speech.Speech"/>
  </service>
</scr:component>
```

It is possible to add activate and deactivate methods on the component with the Activate and Deactivate annotations. If the component wants to be updated for changes in the configuration properties then it can also indicate the modified method with the Modified annotation. For example:

```
@Activate
void open(Map<String,?> properties) { ... }

@Deactivate
void close() { ... }

@Modified
void modified(Map<String,?> properties) { ... }
```

The Activate annotation can also be used on a field or a constructor. When used on a field, the field will be set during activation of the component. When used on a constructor, the constructor will be used to construct the component instances.

```
@Activate
ComponentContext context;

@Activate
public MacSpeech(Map<String,?> properties) { ... }
```

If a component has dependencies on other services then they can be referenced with the Reference annotation that can be applied to a bind method, a field, or a constructor parameter. For a bind method, the defaults for the Reference annotation are:

- The name of the method is used for the name of the reference.
- 1:1 cardinality.
- Static reluctant policy.
- The requested service is the type of the first parameter of the bind method.
- It will infer a default unset method and updated method based on the name of the bind method.

For example:

```
@Reference(cardinality=MULTIPLE, policy=DYNAMIC)
void setLogService( LogService log, Map<String,?> props) { ... }
void unsetLogService( LogService log ) {  ... }
void updatedLogService( Map<String,?> map ) { ...}
```

For a field, the defaults for the Reference annotation are:

- The name of the field is used for the name of the reference.
- 0..n cardinality if the field is a collection. 0..1 cardinality if the field is of type Optional. 1:1 cardinality otherwise.
- Static reluctant policy if the field is not declared volatile. Dynamic reluctant policy if the field is declared volatile.
- The requested service is the type of the field.

For example:

```
@Reference
volatile Collection<LogService> log;
```

For a constructor parameter, the defaults for the Reference annotation are:

- The name of the parameter is used for the name of the reference if method parameter names are included in the class file. Otherwise, a name for the reference must be generated.
- 0..n cardinality if the parameter is a collection. 0..1 cardinality if the parameter is of type Optional. 1:1 cardinality otherwise.
- Static reluctant policy.
- The requested service is the type of the parameter.

For example:

```
@Activate
public MacSpeech(@Reference Collection<LogService> log) { ... }
```

112.8.2 Component Property Types

Component properties can be defined and accessed through a user defined annotation type, called a *component property type*, containing the property names, property types and default values. A component property type allows properties to be defined and accessed in a type safe manner. Component property types can themselves be annotated with the ComponentPropertyType meta-annotation.

The following example shows the definition of a component property type called Config which defines three properties where the name of the property is the name of the method, the type of the property is the return type of the method and the default value for the property is the default value of the method.

```
@ComponentPropertyType
public @interface Config {
  boolean enabled() default true;
  String[] names() default {"a", "b"};
  String topic() default "default/topic";
}
```

Component property types can be used in two ways:

- Component property types can be used to annotate the component implementation class, along side the Component annotation. The annotation usage can specify property values which can be different than the default values declared in the component property type.

  To be used in this way, the component property type must be annotated with the Component-PropertyType meta-annotation so that, at build time, the annotation is recognized as a component property type.
- Component property types can be used as parameter types in the component's constructor and life cycle methods, or as field types for activation fields. The component implementation can use objects of a component property type at runtime to access component property values in a type safe manner.

  To be used in this way, it is recommended the component property type be annotated with the ComponentPropertyType meta-annotation but it is not required.

Both ways define property names, types and values for the component.

The following example shows the component implementation annotated with the example Config component property type which specifies a property value for the component which is different than the default value. The example also shows the activate method taking the example Config component property type as a parameter type and the method implementation accesses component property values by invoking methods on the component property type object.

```
@Component
@Config(names="myapp")
public class MyComponent {
  @Activate
  void activate(Config config) {
    if (config.enabled()) {
      // do something
    }
    for (String name:config.names()) {
      // do something with each name
    }
  }
}
```

If a component implementation needs to access component properties which are not represented by a component property type, it can use a type of Map to receive the properties map in addition to component property types. For example:

```
@Component
public class MyComponent {
  @Activate
  void activate(Config config, Map<String, ?> allProperties) {
    if (config.enabled()) {
      // do something
    }
    if (allProperties.get("other.prop") != null) {
      // do something
    }
  }
}
```

Component property types must be defined as annotation types. This is done for several reasons. First, the limitations on annotation type definitions make them well suited for component property types. The methods must have no parameters and the return types supported are limited to a set which is well suited for component properties. Second, annotation types support default values which is useful for defining the default value of a component property. Finally, as annotations, they can be used to annotate component implementation classes.

At build time, the component property types must be processed to potentially generate property elements in the component description. See *Ordering of Generated Component Properties* on page 297.

At runtime, when SCR needs to provide a component instance an activation object whose type is a component property type, SCR must construct an instance of the component property type whose methods are backed by the values of the component properties for the component instance. This object can then be used to obtain the property values in a type safe manner.

112.8.2.1 Component Property Mapping

Each method of a configuration property type is mapped to a component property. The property name is derived from the method name. Certain common property name characters, such as full stop ('.' \u002E) and hyphen-minus ('-' \u002D) are not valid in Java identifiers. So the name of a method must be converted to its corresponding property name as follows:

- A single dollar sign ('$' \u0024) is removed unless it is followed by:
  - A low line ('_' \u005F) and a dollar sign in which case the three consecutive characters ("$_$") are converted to a single hyphen-minus ('-' \u002D).
  - Another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.
- A single low line ('_' \u005F) is converted into a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.
- If the component property type declares a PREFIX_ field whose value is a compile-time constant String, then the property name is prefixed with the value of the PREFIX_ field.

Table 112.11 contains some name mapping examples.

Table 112.11 Component Property Name Mapping Examples

Component Property Type Method Name	Component Property Name
myProperty143	myProperty143
$new	new
my$$prop	my$prop
dot_prop	dot.prop
_secret	.secret
another__prop	another_prop
three___prop	three_.prop
four_$__prop	four._prop
five_$_prop	five..prop
six$_$prop	six-prop
seven$$_$prop	seven$.prop

However, if the component property type is a *single-element annotation*, see 9.7.3 in [7] *The Java Language Specification, Java SE 8 Edition*, then the property name for the value method is derived from the name of the component property type rather than the name of the method.

In this case, the simple name of the component property type, that is, the name of the class without any package name or outer class name, if the component property type is an inner class, must be converted to the property name as follows:

- When a lower case character is followed by an upper case character, a full stop ('.' \u002E) is inserted between them.
- Each upper case character is converted to lower case.
- All other characters are unchanged.
- If the component property type declares a PREFIX_ field whose value is a compile-time constant String, then the property name is prefixed with the value of the PREFIX_ field.

Table 112.12 contains some mapping examples for the value method.

Table 112.12 *Single-Element Annotation Mapping Examples for* value *Method*

Component Property Type Name	value **Method Component Property Name**
ServiceRanking	service.ranking
Some_Name	some_name
OSGiProperty	osgi.property

If the component property type is a *marker annotation*, see 9.7.2 in [7] *The Java Language Specification, Java SE 8 Edition*, then the property name is derived from the name of the component property type, as is described above for single-element annotations, and the value of the property is Boolean.TRUE. Marker annotations can be used to annotate component implementation classes to set a component property to the value Boolean.TRUE. However, since marker annotations have no methods, they are of no use as parameter types in the component's constructor and life cycle methods, or as field types for activation fields.

The property type can be directly derived from the type of the method. All types supported for annotation elements can be used except for annotation types. Method types of an annotation type or array thereof are not supported. A tool processing the component property types must ignore such methods.

If the method type is Class or Class[], then the property type must be String or String[], respectively, whose values are fully qualified class names in the form returned by the Class.getName() method.

If the method type is an enumeration type or an array thereof, then the property type must be String or String[], respectively, whose values are the names of the enum constants in the form returned by the Enum.name() method.

112.8.2.2 Coercing Component Property Values

When a component property type is used as an activation object type, SCR must create an object that implements the component property type and maps the methods of the component property type to component properties. The name of the method is converted to the property name as described in *Component Property Mapping* on page 294. The property value may need to be coerced to the type of the method. In Table 112.13, the columns are source types, that is, the type of the component property value, and the rows are target types, that is, the method types. The property value is *v*; *number* is a primitive numerical type and *Number* is a wrapper numerical type. An invalid coercion is represented by throw. Such a coercion attempt must result in throwing a Component Exception when the component property type method is called. Any other coercion error, such as parsing a non-numerical string to a number or the inability to coerce a string into a Class or enum object, must be wrapped in a Component Exception and thrown when the component property type method is called.

Table 112.13 Coercion From Property Value to Method Type

target \ source	String	Boolean	Character	*Number*	Collection/array
String	*v*	*v*. toString()	*v*. toString()	*v*. toString()	If *v* has no elements, null; otherwise the first element of *v* is coerced.
boolean	Boolean. parseBoolean(*v*)	*v*. booleanValue()	*v*. charValue() != 0	*v*. doubleValue() != 0	If *v* has no elements, false; otherwise the first element of *v* is coerced.
char	*v*. length() > 0 ? *v*. charAt(0) : 0	*v*. booleanValue() ? 1 : 0	*v*. charValue()	(char) *v*. intValue()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
number	*Number*. parse*Number*(*v*)	*v*. booleanValue() ? 1 : 0	(*number*) *v*. charValue()	*v*. *number*Value()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
Class	Bundle. loadClass(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
EnumType	*EnumType*. valueOf(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
annotation type	throw	throw	throw	throw	throw
array	A single element array is created and *v* is coerced into the single element of the new array.				An array the size of *v* is created and each element of *v* is coerced into the corresponding element of the new array.

Component properties whose names do not map to component property type methods are ignored. If there is no corresponding component property for a component property type method, the component property type method must:

- Return 0 for numerical and char method types.
- Return false for boolean method type.
- Return null for String, Class, and enum.
- Return an empty array for array method types.
- Throw a ComponentException for annotation method types.

112.8.2.3 Standard Component Property Types

Component property types for standard component properties are specified in the org.osgi.service.component.propertytypes package.

The ServiceDescription component property type can be used to add the service.description service property to a component. The ServiceRanking component property type can be used to add the service.ranking service property to a component. The ServiceVendor component property type can be used to add the service.vendor service property to a component. For example, using these component property types as annotations:

```
@Component
@ServiceDescription(”My Acme Service implementation”)
@ServiceRanking(100)
@ServiceVendor("My Corp")
public class MyComponent implements AcmeService {}
```

will result in the following component properties:

```
<property name=”service.description” value=”My Acme Service implementation”/>
```

```
<property name="service.ranking" type="Integer" value="100"/>
<property name="service.vendor" value="My Corp"/>
```

The ExportedService component property type can be used to specify service properties for remote services.

The SatisfyingConditionTarget component property type can be used to specify the target property for a reference to the satisfying condition of a component configuration. See *Satisfying Condition* on page 264.

112.8.3 Ordering of Generated Component Properties

The Component annotation contains two ways to define component properties via the property and properties elements. See *Property and Properties Elements* on page 268. If Component Annotations are used to describe the component, then any component property types used as the type of an activation object or used to annotate the component implementation class must also be processed since component property types can be used to define component property values as well. See *Component Property Types* on page 292. A tool processing the Component Annotations and the component property types must write the defined component properties into the generated component description in the following order.

1. Properties defined through component property types used as the type of an activation object.

   If any of the referenced component property types have methods with defaults, then the generated component description must include a property element for each such method with the property name mapped from the method name, the property type mapped from the method type, and the property value set to the method's default value. See *Component Property Mapping* on page 294. The generated property elements must be added to the component description by processing the component property types used as the type of an activation object in the following order:

   a. The component property types used as parameters to the constructor.
   b. The component property types used as activation fields. The fields are processed in lexicographical order, using String.compareTo, of the field names.
   c. The component property types used as parameters to the activate method.
   d. The component property types used as parameters to the modified method.
   e. The component property types used as parameters to the deactivate method.

   If a method has more than one component property type parameter, the component property types are processed in the order of the method parameters.

   For component property type methods without a default value or with a default value of an empty array, a property element must not be generated.
2. Properties defined through component property types annotating the component implementation class.

   The generated component description must include a property element for each such method with the property name mapped from the method name, the property type mapped from the method type, and the property value set to the method's value. See *Component Property Mapping* on page 294. The generated property elements must be added to the component description by processing the component property types annotating the component implementation class in the order that the annotations appear in the component implementation's class file. However, the order of the RuntimeVisibleAnnotations and RuntimeInvisibleAnnotations attributes in the class file is unspecified by [6] *The Java Virtual Machine Specification, Java SE 8 Edition* so care must be taken when using component property types of different RetentionPolicy that have method names in common.

   For component property type methods with a value of an empty array, a property element must not be generated.

3. property element of the Component annotation.
4. properties element of the Component annotation.

This means that the properties defined through component property types are declared first in the generated component description, followed by all properties defined through the property element of the Component annotation and finally the properties entries defined through the properties element of the Component annotation.

Since property values defined later in the component description override property values defined earlier in the component description, this means that property values defined in properties element of the Component annotation can override property values defined in property element of the Component annotation which can override values defined by values in the component property types.

112.9 Service Component Runtime

Service Component Runtime (SCR) is the actor that manages the components and their life cycle and allows introspection of the components.

112.9.1 Relationship to OSGi Framework

SCR must have access to the Bundle Context of any bundle that contains a component. SCR needs access to the Bundle Context for the following reasons:

- To be able to register and get services on behalf of a bundle with components.
- To interact with the Configuration Admin on behalf of a bundle with components.
- To provide a component its Bundle Context when the Component Context getBundleContext method is called.

SCR should use the Bundle.getBundleContext() method to obtain the Bundle Context reference.

112.9.2 Starting and Stopping SCR

When SCR is implemented as a bundle, any component configurations activated by SCR must be deactivated when the SCR bundle is stopped. When the SCR bundle is started, it must process any components that are declared in bundles that are started. This includes bundles which are started and are awaiting lazy activation.

112.9.3 Logging Messages

When SCR must log a message to the Log Service, it must use a Logger named for the component implementation class and associated with the bundle declaring the component. To obtain the Logger object, SCR must call the LoggerFactory.getLogger(Bundle bundle, String name, Class loggerType) method passing the bundle declaring the component as the first argument and the fully qualified name of the component implementation class as the second argument. If SCR cannot know the component implementation class name, because the error is not associated with a component or the error occurred before the component description is processed, then SCR must use the bundle's Root Logger, that is, the Logger named ROOT.

112.9.4 Locating Component Methods and Fields

SCR will need to locate activate, deactivate, modified, bind, updated, and unbind methods as well as fields in a component instance. These members will be located, and called or modified, using reflection. The declared members of each class in the component implementation class's hierarchy are examined for a suitable member. If a suitable member is found in a class, and it is accessible to the component implementation class, then that member must be used. If suitable members are found in a class but none of the suitable members are accessible by the component implementation class, then the search for suitable members terminates with no suitable member having been located. If no suitable members are found in a class, the search continues in the superclass.

Only members that are accessible to the component implementation class will be used. If the member has the public or protected access modifier, then access is permitted. Otherwise, if the member has the private access modifier, then access is permitted only if the member is declared in the component implementation class. Otherwise, if the member has default access, also known as package private access, then access is permitted only if the member is declared in the component implementation class or if the member is declared in a superclass and all classes in the hierarchy from the component implementation class to the superclass, inclusive, are in the same package and loaded by the same class loader.

It is recommended that these members should not be declared with the public access modifier so that they do not appear as public members on the component instance when it is used as a service object. Having these members declared public allows any code to call or access the members with reflection, even if a Security Manager is installed. These members are generally intended to only be called or modified by SCR.

112.9.5 Bundle Activator Interaction

A bundle containing components may also declare a Bundle Activator. Such a bundle may also be marked for lazy activation. Since components are activated by SCR and Bundle Activators are called by the OSGi Framework, a bundle using both components and a Bundle Activator must take care. The Bundle Activator's start method must not rely upon SCR having activated any of the bundle's components. However, the components can rely upon the Bundle Activator's start method having been called. That is, there is a *happens-before* relationship between the Bundle Activator's start method being run and the components being activated.

112.9.6 Introspection

SCR provides an introspection API for examining the runtime state of the components in bundles processed by SCR. SCR must register a ServiceComponentRuntime service upon startup. The Service Component Runtime service provides methods to inspect the component descriptions and component configurations as well as inspect and modify the enabled state of components. The service uses *Data Transfer Objects (DTO)* as parameters and return values. The rules for Data Transfer Objects are specified in *OSGi Core Release 8*.

The Service Component Runtime service provides the following methods.

- getComponentDescriptionDTOs(Bundle...) - For each specified bundle, if the bundle is active and processed by SCR, the returned collection will contain a ComponentDescriptionDTO for each valid component description in the bundle.
- getComponentDescriptionDTO(Bundle,String) - If the specified bundle is active and processed by SCR, and the specified bundle contains a valid component description with the specified name, the method will return a ComponentDescriptionDTO for the component description.
- getComponentConfigurationDTOs(ComponentDescriptionDTO) - If the specified ComponentDescriptionDTO represents a valid component description from an active bundle processed by SCR, the returned collection will contain a ComponentConfigurationDTO for each component configuration of the component.
- isComponentEnabled(ComponentDescriptionDTO) - Returns true if the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, and the component is enabled. Otherwise, the method returns false.
- enableComponent(ComponentDescriptionDTO) - If the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, the component is enabled. This method must return after changing the enabled state of the specified component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call. The method returns a Promise that will

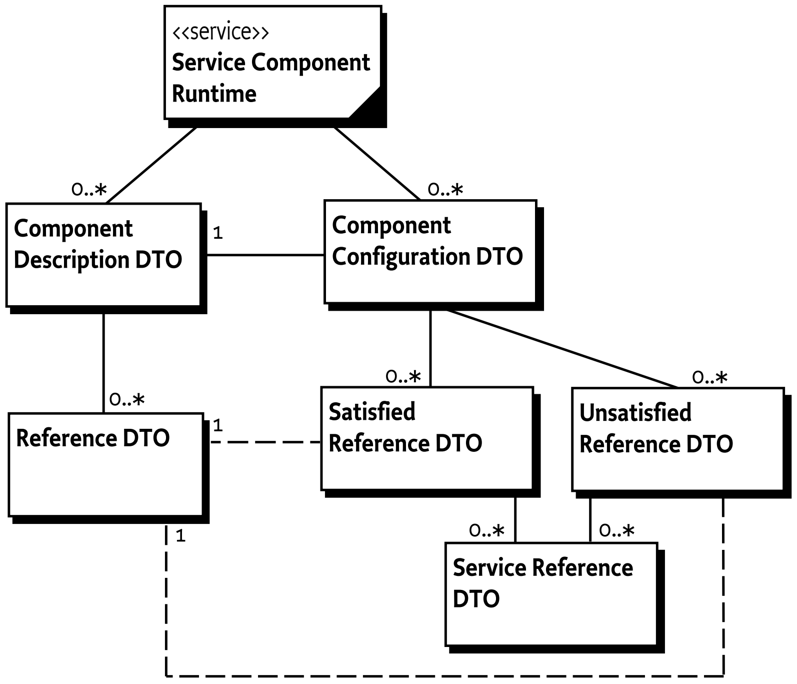
be resolved when the actions that result from changing the enabled state of the specified component have completed.

- disableComponent(ComponentDescriptionDTO) - If the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, the component is disabled. This method must return after changing the enabled state of the specified component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call. The method returns a Promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed.

The runtime state of the components can change at any time. So any information returned by these methods only provides a snapshot of the state at the time of the method call.

There are a number of DTOs available via the Service Component Runtime service.

Figure 112.6 *Service Component Runtime DTOs*



<<service>>
Service Component Runtime
0..*
0..*
Component Description DTO
1
Component Configuration DTO
0..*
0..*
0..*
Reference DTO
1
Satisfied Reference DTO
Unsatisfied Reference DTO
1
0..*
0..*
Service Reference DTO

The two main DTOs are ComponentDescriptionDTO, which represents a component description, and ComponentConfigurationDTO, which represents a component configuration. The Component Description DTO contains an array of ReferenceDTO objects which represent each declared reference in the component description. The Component Configuration DTO contains an array of SatisfiedReferenceDTO objects and an array of UnsatisfiedReferenceDTO objects. A Satisfied Reference DTO represents a satisfied reference of the component configuration and an Unsatisfied Reference DTO represents an unsatisfied reference of the component configuration. The Component Configuration DTO for a satisfied component configuration must contain no Unsatisfied Reference DTOs. The Component Configuration DTO for an unsatisfied component configuration may contain some Satisfied Reference DTOs and some Unsatisfied Reference DTOs. This information can be used to diagnose why the component configuration is not satisfied.

SCR must register the ServiceComponentRuntime service with the service.changecount service property. See org.osgi.framework.Constants.SERVICE_CHANGECOUNT in *OSGi Core Release 8*. Whenever the Service Component Runtime DTOs available from the ServiceComponentRuntime service change, SCR modify the service.changecount service property with an updated change count value. This allows interested parties to be notified of changes to the DTOs by observing Service Events of type MODIFIED for the ServiceComponentRuntime service.

112.9.7 Capabilities

SCR must provide the following capabilities.

- A capability in the osgi.extender namespace declaring an extender with the name COMPONENT_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.component package. For example:

  ```
  Provide-Capability: osgi.extender;
      osgi.extender="osgi.component";
      version:Version="1.5";
      uses:="org.osgi.service.component"
  ```

  This capability must follow the rules defined for the *osgi.extender Namespace* on page 707.

  A bundle that contains service components should require the osgi.extender capability from SCR. This requirement will wire the bundle to the SCR implementation and ensure that SCR is using the same org.osgi.service.component package as the bundle if the bundle uses that package.

  ```
  Require-Capability: osgi.extender;
    filter:="(&(osgi.extender=osgi.component)(version>=1.5)(!(version>=2.0)))"
  ```

  The RequireServiceComponentRuntime annotation can be used to require this capability. The Component annotation is meta-annotated with this annotation.

  SCR must only process a bundle's service components if one of the following is true:
  - The bundle's wiring has a required wire for at least one osgi.extender capability with the name osgi.component and the first of these required wires is wired to SCR.
  - The bundle's wiring has no required wire for an osgi.extender capability with the name osgi.component.

  Otherwise, SCR must not process the bundle's service components.
- A capability in the osgi.service namespace representing the ServiceComponentRuntime service. This capability must also declare a uses constraint for the org.osgi.service.component.runtime package. For example:

  ```
  Provide-Capability: osgi.service;
      objectClass:List<String>=
        "org.osgi.service.component.runtime.ServiceComponentRuntime";
      uses:="org.osgi.service.component.runtime"
  ```

  This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

112.9.8 Locating the True Condition Service

SCR must locate the True Condition service. This can be done using the Bundle Context of SCR itself. However, if SCR is unable to locate the True Condition service using the Bundle Context of SCR itself, it may retry using the Bundle Context of the system bundle. This retry to locate the True Condition service can succeed when a service hook implementation is in use that has not been updated to support the Condition Service Specification by providing visibility of the True Condition service to all bundles.

If SCR is unable to locate the True Condition service, which can occur if SCR is running on an older OSGi Framework which does not support the Condition Service Specification and register the True

Condition service, then SCR must not augment component descriptions to add the implicit reference for a satisfying condition. See *Satisfying Condition* on page 264.

SCR may use the already located True Condition service to satisfy any component configuration's reference to the True Condition service.

112.10 Security

When Java permissions are enabled, SCR must perform the following security procedures.

112.10.1 Service Permissions

Declarative services are built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish, find or bind services.

If a component specifies a service, then component configurations for the component cannot be satisfied unless the component's bundle has `ServicePermission[<provides>, REGISTER]` for each provided interface specified for the service.

If a component's reference does not specify optional cardinality, the reference cannot be satisfied unless the component's bundle has `ServicePermission[<interface>, GET]` for the specified interface in the reference. If the reference specifies optional cardinality but the component's bundle does not have `ServicePermission[<interface>, GET]` for the specified interface in the reference, no service must be bound for this reference.

If a component is a factory component, then the above Service Permission checks still apply. But the component's bundle is not required to have `ServicePermission[ComponentFactory, REGISTER]` as the Component Factory service is registered by SCR.

SCR must have `ServicePermission[ServiceComponentRuntime, REGISTER]` permission to register the `ServiceComponentRuntime` service. Administrative bundles wishing to use the `ServiceComponentRuntime` service must have `ServicePermission[ServiceComponentRuntime, GET]` permission. In general, this permission should only be granted to administrative bundles to limit access to the potentially intrusive methods provided by this service.

112.10.2 Required Admin Permission

SCR requires `AdminPermission[*,CONTEXT]` because it needs access to the bundle's Bundle Context object with the `Bundle.getBundleContext()` method.

112.10.3 Using hasPermission

SCR does all publishing, finding and binding of services on behalf of the component using the Bundle Context of the component's bundle. This means that normal stack-based permission checks will check SCR and not the component's bundle. Since SCR is registering and getting services on behalf of a component's bundle, SCR must call the `Bundle.hasPermission` method to validate that a component's bundle has the necessary permission to register or get a service.

112.10.4 Configuration Multi-Locations and Regions

SCR must ensure a bundle has the proper `ConfigurationPermission` for a Configuration used by its components when the Configuration has a multi-location. See *Using Multi-Locations* on page 83 for more information on multi-locations and *Regions* on page 84 for more information on regions. If a bundle does not have the necessary permission for a multi-location Configuration, then SCR must act as if the Configuration does not exist for the bundle.

112.11 Component Description Schema

This XML Schema defines the component description grammar.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.5.0"
    targetNamespace="http://www.osgi.org/xmlns/scr/v1.5.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.5.0">

    <annotation>
        <documentation xml:lang="en">
            This is the XML Schema for component descriptions used by
            the Service Component Runtime (SCR). Component description
            documents may be embedded in other XML documents. SCR will
            process all XML documents listed in the Service-Component
            manifest header of a bundle. XML documents containing
            component descriptions may contain a single, root component
            element or one or more component elements embedded in a
            larger document. Use of the namespace for component
            descriptions is mandatory. The attributes and subelements
            of a component element are always unqualified.
        </documentation>
    </annotation>
    <element name="component" type="scr:Tcomponent" />
    <complexType name="Tcomponent">
        <sequence>
            <annotation>
                <documentation xml:lang="en">
                    Implementations of SCR must not require component
                    descriptions to specify the subelements of the component
                    element in the order as required by the schema. SCR
                    implementations must allow other orderings since
                    arbitrary orderings do not affect the meaning of the
                    component description. Only the relative ordering of
                    property and properties element have meaning.
                </documentation>
            </annotation>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="property" type="scr:Tproperty" />
                <element name="properties" type="scr:Tproperties" />
            </choice>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="factory-property" type="scr:Tproperty" />
                <element name="factory-properties" type="scr:Tproperties" />
            </choice>
            <element name="service" type="scr:Tservice" minOccurs="0"
                maxOccurs="1" />
            <element name="reference" type="scr:Treference"
                minOccurs="0" maxOccurs="unbounded" />
            <element name="implementation" type="scr:Timplementation" />
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="enabled" type="boolean" default="true"
            use="optional" />
        <attribute name="name" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the class attribute of the nested implementation
                    element. If multiple component elements use the same
                    value for the class attribute of their nested
                    implementation element, then using the default value
                    for this attribute will result in duplicate names.
                    In this case, this attribute must be specified with
                    a unique value.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="factory" type="string" use="optional" />
        <attribute name="immediate" type="boolean" use="optional" />
        <attribute name="configuration-policy"
            type="scr:Tconfiguration-policy" default="optional" use="optional" />
        <attribute name="activate" type="token" use="optional"
            default="activate" />
        <attribute name="deactivate" type="token" use="optional"
            default="deactivate" />
        <attribute name="modified" type="token" use="optional" />
        <attribute name="configuration-pid" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the name attribute of this element.
                </documentation>
            </annotation>
            <simpleType>
                <restriction>
                    <simpleType>
                        <list itemType="token" />
                    </simpleType>
                    <minLength value="1" />
                </restriction>
            </simpleType>
        </attribute>
        <attribute name="activation-fields" use="optional">
            <simpleType>
                <restriction>
                    <simpleType>
                        <list itemType="token" />
                    </simpleType>
                    <minLength value="1" />
                </restriction>
            </simpleType>
        </attribute>
        <attribute name="init" type="unsignedByte" default="0"
            use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Timplementation">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="class" type="token" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tproperty">
        <simpleContent>
            <extension base="string">
                <attribute name="name" type="string" use="required" />
                <attribute name="value" type="string" use="optional" />
                <attribute name="type" type="scr:Tproperty_type"
                    default="String" use="optional" />
                <anyAttribute processContents="lax" />
            </extension>
        </simpleContent>
    </complexType>
    <complexType name="Tproperties">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="entry" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tservice">
        <sequence>
            <element name="provide" type="scr:Tprovide" minOccurs="1"
                maxOccurs="unbounded" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="scope" type="scr:Tservice_scope" default="singleton"
```

```
            use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tprovide">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="interface" type="token" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Treference">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the interface attribute of this element. If multiple
                    instances of this element within a component element
                    use the same value for the interface attribute, then
                    using the default value for this attribute will result
                    in duplicate names. In this case, this attribute
                    must be specified with a unique value.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="interface" type="token" use="required" />
        <attribute name="cardinality" type="scr:Tcardinality"
            default="1..1" use="optional" />
        <attribute name="policy" type="scr:Tpolicy" default="static"
            use="optional" />
        <attribute name="policy-option" type="scr:Tpolicy-option"
            default="reluctant" use="optional" />
        <attribute name="target" type="string" use="optional" />
        <attribute name="bind" type="token" use="optional" />
        <attribute name="unbind" type="token" use="optional" />
        <attribute name="updated" type="token" use="optional" />
        <attribute name="scope" type="scr:Treference_scope" default="bundle"
            use="optional" />
        <attribute name="field" type="token" use="optional" />
        <attribute name="field-option" type="scr:Tfield-option" default="replace"
            use="optional" />
        <attribute name="field-collection-type" type="scr:Tfield-collection-type"
            default="service" use="optional" />
        <attribute name="parameter" type="unsignedByte" use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>
    <simpleType name="Tproperty_type">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="Float" />
            <enumeration value="Integer" />
            <enumeration value="Byte" />
            <enumeration value="Character" />
            <enumeration value="Boolean" />
            <enumeration value="Short" />
        </restriction>
    </simpleType>
    <simpleType name="Tcardinality">
        <restriction base="string">
            <enumeration value="0..1" />
            <enumeration value="0..n" />
            <enumeration value="1..1" />
            <enumeration value="1..n" />
        </restriction>
    </simpleType>
    <simpleType name="Tpolicy">
        <restriction base="string">
            <enumeration value="static" />
            <enumeration value="dynamic" />
        </restriction>
    </simpleType>
    <simpleType name="Tpolicy-option">
        <restriction base="string">
            <enumeration value="reluctant" />
            <enumeration value="greedy" />
        </restriction>
    </simpleType>
    <simpleType name="Tconfiguration-policy">
        <restriction base="string">
            <enumeration value="optional" />
            <enumeration value="require" />
            <enumeration value="ignore" />
        </restriction>
    </simpleType>
    <simpleType name="Tservice_scope">
        <restriction base="string">
            <enumeration value="singleton" />
            <enumeration value="bundle" />
            <enumeration value="prototype" />
        </restriction>
    </simpleType>
    <simpleType name="Treference_scope">
        <restriction base="string">
            <enumeration value="bundle" />
            <enumeration value="prototype" />
            <enumeration value="prototype_required" />
        </restriction>
    </simpleType>
    <simpleType name="Tfield-option">
        <restriction base="string">
            <enumeration value="replace" />
            <enumeration value="update" />
        </restriction>
    </simpleType>
    <simpleType name="Tfield-collection-type">
        <restriction base="string">
            <enumeration value="service" />
            <enumeration value="properties" />
            <enumeration value="reference" />
            <enumeration value="serviceobjects" />
            <enumeration value="tuple" />
        </restriction>
    </simpleType>
    <attribute name="must-understand" type="boolean">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents
                to require that the document consumer understand the
                extension. This attribute must be qualified when used.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

SCR must not require component descriptions to specify the elements in the order required by the schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of `property`, `properties` and `reference` elements have meaning for overriding previously set property values.

The schema is also available in digital form from [5] *OSGi XML Schemas*.

112.12 org.osgi.service.component

Service Component Package Version 1.5.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.component; version="[1.5,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.component; version="[1.5,1.6)"`

112.12.1 Summary

- `AnyService` - A marker type whose name is used in the `interface` attribute of a `reference` element in a component description to indicate that the type of the service for a reference is not specified and can thus be any service type.
- `ComponentConstants` - Defines standard names for Service Component constants.
- `ComponentContext` - A Component Context object is used by a component instance to interact with its execution context including locating services by reference name.
- `ComponentException` - Unchecked exception which may be thrown by Service Component Runtime.
- `ComponentFactory` - When a component is declared with the `factory` attribute on its `component` element, Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.
- `ComponentInstance` - A ComponentInstance encapsulates a component instance of an activated component configuration.
- `ComponentServiceObjects` - Allows multiple service objects for a service to be obtained.

112.12.2 public final class AnyService

A marker type whose name is used in the `interface` attribute of a `reference` element in a component description to indicate that the type of the service for a reference is not specified and can thus be any service type.

When specifying this marker type in the `interface` attribute of a `reference` element in a component description:

- The service type of the reference member or parameter must be `java.lang.Object` so that any service object can be provided.
- The `target` attribute of the `reference` element must be specified to constrain the target services.

For example:

```
@Reference(service = AnyService.class, target = "(osgi.jakartars.extension=true)")
List<Object> extensions;
```

Since 1.5

112.12.3 public interface ComponentConstants

Defines standard names for Service Component constants.

Provider Type Consumers of this API must not implement this type

112.12.3.1 public static final String COMPONENT_CAPABILITY_NAME = "osgi.component"

Capability name for Service Component Runtime.

Used in `Provide-Capability` and `Require-Capability` manifest headers with the `osgi.extender` namespace. For example:

```
Require-Capability: osgi.extender;
```

```
filter:="(&(osgi.extender=osgi.component)(version>=1.5)(!(version>=2.0)))"
```

Since 1.3

112.12.3.2 public static final String COMPONENT_FACTORY = "component.factory"

A service registration property for a Component Factory that contains the value of the `factory` attribute. The value of this property must be of type `String`.

112.12.3.3 public static final String COMPONENT_ID = "component.id"

A component property that contains the generated id for a component configuration. The value of this property must be of type `Long`.

The value of this property is assigned by Service Component Runtime when a component configuration is created. Service Component Runtime assigns a unique value that is larger than all previously assigned values since Service Component Runtime was started. These values are NOT persistent across restarts of Service Component Runtime.

112.12.3.4 public static final String COMPONENT_NAME = "component.name"

A component property for a component configuration that contains the name of the component as specified in the `name` attribute of the `component` element. The value of this property must be of type `String`.

112.12.3.5 public static final String COMPONENT_SPECIFICATION_VERSION = "1.5"

Compile time constant for the Specification Version of Declarative Services.

Used in `Version` and `Requirement` annotations. The value of this compile time constant will change when the specification version of Declarative Services is updated.

Since 1.4

112.12.3.6 public static final int DEACTIVATION_REASON_BUNDLE_STOPPED = 6

The component configuration was deactivated because the bundle was stopped.

Since 1.1

112.12.3.7 public static final int DEACTIVATION_REASON_CONFIGURATION_DELETED = 4

The component configuration was deactivated because its configuration was deleted.

Since 1.1

112.12.3.8 public static final int DEACTIVATION_REASON_CONFIGURATION_MODIFIED = 3

The component configuration was deactivated because its configuration was changed.

Since 1.1

112.12.3.9 public static final int DEACTIVATION_REASON_DISABLED = 1

The component configuration was deactivated because the component was disabled.

Since 1.1

112.12.3.10 public static final int DEACTIVATION_REASON_DISPOSED = 5

The component configuration was deactivated because the component was disposed.

Since 1.1

112.12.3.11 public static final int DEACTIVATION_REASON_REFERENCE = 2

The component configuration was deactivated because a reference became unsatisfied.

Since 1.1

112.12.3.12 public static final int DEACTIVATION_REASON_UNSPECIFIED = 0

The reason the component configuration was deactivated is unspecified.

Since 1.1

112.12.3.13 public static final String REFERENCE_NAME_SATISFYING_CONDITION = "osgi.ds.satisfying.condition"

Reference name for a component's satisfying condition.

Since 1.5

112.12.3.14 public static final String REFERENCE_TARGET_SUFFIX = ".target"

The suffix for the target property of a reference. These properties contain the filter to select the target services for a reference. The value of a target property must be of type `String`.

112.12.3.15 public static final String SERVICE_COMPONENT = "Service-Component"

Manifest header specifying the XML documents within a bundle that contain the bundle's Service Component descriptions.

The attribute value may be retrieved from the `Dictionary` object returned by the `Bundle.getHeaders` method.

112.12.4 public interface ComponentContext

A Component Context object is used by a component instance to interact with its execution context including locating services by reference name. Each component instance has a unique Component Context.

A component instance may obtain its Component Context object through its activate, modified, and deactivate methods.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.4.1 public void disableComponent(String name)

name The name of a component.

□ Disables the specified component name. The specified component name must be in the same bundle as this component.

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

112.12.4.2 public void enableComponent(String name)

name The name of a component or `null` to indicate all components in the bundle.

□ Enables the specified component name. The specified component name must be in the same bundle as this component.

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

112.12.4.3 public BundleContext getBundleContext()

□ Returns the `BundleContext` of the bundle which declares this component.

Returns The `BundleContext` of the bundle declares this component.

112.12.4.4 public ComponentInstance<S> getComponentInstance()

Type Parameters <S>

□ Returns the Component Instance object for the component instance associated with this Component Context.

Returns The Component Instance object for the component instance.

112.12.4.5 public Dictionary<String, Object> getProperties()

□ Returns the component properties for this Component Context.

Returns The properties for this Component Context. The Dictionary is read only and cannot be modified.

112.12.4.6 public ServiceReference<?> getServiceReference()

□ If the component instance is registered as a service using the service element, then this method returns the service reference of the service provided by this component instance.

This method will return null if the component instance is not registered as a service.

Returns The ServiceReference object for the component instance or null if the component instance is not registered as a service.

112.12.4.7 public Bundle getUsingBundle()

□ If the component instance is registered as a service using the servicescope="bundle" or servicescope="prototype" attribute, then this method returns the bundle using the service provided by the component instance.

This method will return null if:

- The component instance is not a service, then no bundle can be using it as a service.
- The component instance is a service but did not specify the servicescope="bundle" or servicescope="prototype" attribute, then all bundles using the service provided by the component instance will share the same component instance.
- The service provided by the component instance is not currently being used by any bundle.

Returns The bundle using the component instance as a service or null.

112.12.4.8 public S locateService(String name)

Type Parameters <S>

name The name of a reference as specified in a reference element in this component's description.

□ Returns the service object for the specified reference name.

If the cardinality of the reference is 0..n or 1..n and multiple services are bound to the reference, the service whose ServiceReference is first in the ranking order is returned. See ServiceReference.compareTo(Object).

Returns A service object for the referenced service or null if the reference cardinality is 0..1 or 0..n and no bound service is available.

Throws ComponentException– If Service Component Runtime catches an exception while activating the bound service.

112.12.4.9 public S locateService(String name, ServiceReference<S> reference)

Type Parameters <S>

<S> Type of Service.

name The name of a reference as specified in a reference element in this component's description.

reference The ServiceReference to a bound service. This must be a ServiceReference provided to the component via the bind or unbind method for the specified reference name.

□ Returns the service object for the specified reference name and ServiceReference.

Returns A service object for the referenced service or null if the specified ServiceReference is not a bound service for the specified reference name.

Throws ComponentException– If Service Component Runtime catches an exception while activating the bound service.

112.12.4.10 public Object[] locateServices(String name)

name The name of a reference as specified in a reference element in this component's description.

□ Returns the service objects for the specified reference name.

Returns An array of service objects for the referenced service or null if the reference cardinality is 0..1 or 0..n and no bound service is available. If the reference cardinality is 0..1 or 1..1 and a bound service is available, the array will have exactly one element. There is no guarantee that the service objects in the array will be in any specific order.

Throws ComponentException– If Service Component Runtime catches an exception while activating a bound service.

112.12.5 public class ComponentException extends RuntimeException

Unchecked exception which may be thrown by Service Component Runtime.

112.12.5.1 public ComponentException(String message, Throwable cause)

message The message for the exception.

cause The cause of the exception. May be null.

□ Construct a new ComponentException with the specified message and cause.

112.12.5.2 public ComponentException(String message)

message The message for the exception.

□ Construct a new ComponentException with the specified message.

112.12.5.3 public ComponentException(Throwable cause)

cause The cause of the exception. May be null.

□ Construct a new ComponentException with the specified cause.

112.12.5.4 public Throwable getCause()

□ Returns the cause of this exception or null if no cause was set.

Returns The cause of this exception or null if no cause was set.

112.12.5.5 public Throwable initCause(Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws IllegalArgumentException– If the specified cause is this exception.

IllegalStateException– If the cause of this exception has already been set.

112.12.6 public interface ComponentFactory<S>

<S> Type of Service

When a component is declared with the `factory` attribute on its `component` element, Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.6.1 public ComponentInstance<S> newInstance(Dictionary<String, ?> properties)

properties Additional properties for the component configuration or `null` if there are no additional properties.

□ Create and activate a new component configuration. Additional properties may be provided for the component configuration.

Returns A `ComponentInstance` object encapsulating the component instance of the component configuration. The component configuration has been activated and, if the component specifies a `service` element, the component instance has been registered as a service.

Throws `ComponentException`– If Service Component Runtime is unable to activate the component configuration.

112.12.7 public interface ComponentInstance<S>

<S> Type of Service

A ComponentInstance encapsulates a component instance of an activated component configuration. ComponentInstances are created whenever a component configuration is activated.

ComponentInstances are never reused. A new ComponentInstance object will be created when the component configuration is activated again.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.7.1 public void dispose()

□ Dispose of the component configuration for this component instance. The component configuration will be deactivated. If the component configuration has already been deactivated, this method does nothing.

112.12.7.2 public S getInstance()

□ Returns the component instance of the activated component configuration.

Returns The component instance or `null` if the component configuration has been deactivated.

112.12.8 public interface ComponentServiceObjects<S>

<S> Type of Service

Allows multiple service objects for a service to be obtained.

A component instance can receive a `ComponentServiceObjects` object via a reference that is typed `ComponentServiceObjects`.

For services with prototype scope, multiple service objects for the service can be obtained. For services with singleton or bundle scope, only one, use-counted service object is available.

Any unreleased service objects obtained from this `ComponentServiceObjects` object are automatically released by Service Component Runtime when the service becomes unbound.

See Also ServiceObjects

Since 1.3

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.8.1 public S getService()

□ Returns a service object for the associated service.

This method will always return null when the associated service has been become unbound.

Returns A service object for the associated service or null if the service is unbound, the customized service object returned by a ServiceFactory does not implement the classes under which it was registered or the ServiceFactory threw an exception.

Throws IllegalStateException– If the component instance that received this ComponentServiceObjects object has been deactivated.

See Also ungetService(Object)

112.12.8.2 public ServiceReference<S> getServiceReference()

□ Returns the ServiceReference for the service associated with this ComponentServiceObjects object.

Returns The ServiceReference for the service associated with this ComponentServiceObjects object.

112.12.8.3 public void ungetService(S service)

service A service object previously provided by this ComponentServiceObjects object.

□ Releases a service object for the associated service.

The specified service object must no longer be used and all references to it should be destroyed after calling this method.

Throws IllegalStateException– If the component instance that received this ComponentServiceObjects object has been deactivated.

IllegalArgumentException– If the specified service object was not provided by this ComponentServiceObjects object.

See Also getService()

112.13 org.osgi.service.component.annotations

Service Component Annotations Package Version 1.5.

This package is not used at runtime. Annotated classes are processed by tools to generate Component Descriptions which are used at runtime.

112.13.1 Summary

- Activate - Identify the annotated member as part of the activation of a Service Component.
- CollectionType - Collection types for the Reference annotation.
- Component - Identify the annotated class as a Service Component.
- ComponentPropertyType - Identify the annotated annotation as a Component Property Type.
- ConfigurationPolicy - Configuration Policy for the Component annotation.
- Deactivate - Identify the annotated method as the deactivate method of a Service Component.
- FieldOption - Field options for the Reference annotation.

- Modified - Identify the annotated method as the modified method of a Service Component.
- Reference - Identify the annotated member or parameter as a reference of a Service Component.
- ReferenceCardinality - Cardinality for the Reference annotation.
- ReferencePolicy - Policy for the Reference annotation.
- ReferencePolicyOption - Policy option for the Reference annotation.
- ReferenceScope - Reference scope for the Reference annotation.
- RequireServiceComponentRuntime - This annotation can be used to require the Service Component Runtime to process Declarative Services components.
- ServiceScope - Service scope for the Component annotation.

112.13.2 @Activate

Identify the annotated member as part of the activation of a Service Component.

When this annotation is applied to a:

- Method - The method is the activate method of the Component.
- Constructor - The constructor will be used to construct the Component and can be called with activation objects and bound services as parameters.
- Field - The field will contain an activation object of the Component. The field must be set after the constructor is called and before calling any other method on the fully constructed component instance. That is, there is a *happens-before* relationship between the field being set and calling any method on the fully constructed component instance such as the activate method.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also The init, activate, and activation-fields attributes of the component element of a Component Description.

Since 1.1

Retention CLASS

Target METHOD, FIELD, CONSTRUCTOR

112.13.3 enum CollectionType

Collection types for the Reference annotation.

Since 1.4

112.13.3.1 SERVICE

The service collection type is used to indicate the collection holds the bound service objects.

This is the default collection type.

112.13.3.2 REFERENCE

The reference collection type is used to indicate the collection holds Service References for the bound services.

112.13.3.3 SERVICEOBJECTS

The serviceobjects collection type is used to indicate the collection holds Component Service Objects for the bound services.

112.13.3.4 PROPERTIES

The properties collection type is used to indicate the collection holds unmodifiable Maps containing the service properties of the bound services.

The Maps must implement Comparable with the compareTo method comparing service property maps using an ordering which is the same as the natural ordering of ServiceReferences as specified in ServiceReference.compareTo.

112.13.3.5 TUPLE

The tuple collection type is used to indicate the collection holds unmodifiable Map.Entries whose key is an unmodifiable Map containing the service properties of the bound service, as specified in PROPERTIES, and whose value is the bound service object.

The Map.Entries must implement Comparable with the compareTo method comparing service property maps using an ordering which is the same as the natural ordering of ServiceReferences as specified in ServiceReference.compareTo.

112.13.3.6 public String toString()

112.13.3.7 public static CollectionType valueOf(String name)

112.13.3.8 public static CollectionType[] values()

112.13.4 @Component

Identify the annotated class as a Service Component.

The annotated class is the implementation class of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also The component element of a Component Description.

Retention CLASS

Target TYPE

112.13.4.1 String name default ""

□ The name of this Component.

If not specified, the name of this Component is the fully qualified type name of the class being annotated.

See Also The name attribute of the component element of a Component Description.

112.13.4.2 Class<?>[] service default {}

□ The types under which to register this Component as a service.

If no service should be registered, the empty value {} must be specified.

If not specified, the service types for this Component are all the *directly* implemented interfaces of the class being annotated.

See Also The service element of a Component Description.

112.13.4.3 String factory default ""

□ The factory identifier of this Component. Specifying a factory identifier makes this Component a Factory Component.

If not specified, the default is that this Component is not a Factory Component.

See Also The factory attribute of the component element of a Component Description.

112.13.4.4 boolean servicefactory default false

□ Declares whether this Component uses the OSGi ServiceFactory concept and each bundle using this Component's service will receive a different component instance.

This element is ignored when the scope() element does not have the default value. If true, this Component uses bundle service scope. If false or not specified, this Component uses singleton service scope. If the factory() element is specified or the immediate() element is specified with true, this element can only be specified with false.

See Also The scope attribute of the service element of a Component Description.

Deprecated Since 1.3. Replaced by scope().

112.13.4.5 boolean enabled default true

□ Declares whether this Component is enabled when the bundle declaring it is started.

If true or not specified, this Component is enabled. If false, this Component is disabled.

See Also The enabled attribute of the component element of a Component Description.

112.13.4.6 boolean immediate default false

□ Declares whether this Component must be immediately activated upon becoming satisfied or whether activation should be delayed.

If true, this Component must be immediately activated upon becoming satisfied. If false, activation of this Component is delayed. If this property is specified, its value must be false if the factory() property is also specified or must be true if the service() property is specified with an empty value.

If not specified, the default is false if the factory() property is specified or the service() property is not specified or specified with a non-empty value and true otherwise.

See Also The immediate attribute of the component element of a Component Description.

112.13.4.7 String[] property default {}

□ Properties for this Component.

Each property string is specified as "name=value". The type of the property value can be specified in the name as name:type=value. The type must be one of the property types supported by the type attribute of the property element of a Component Description.

To specify a property with multiple values, use multiple name, value pairs. For example, {"foo=bar", "foo=baz"}.

See Also The property element of a Component Description.

112.13.4.8 String[] properties default {}

□ Property entries for this Component.

Specifies the name of an entry in the bundle whose contents conform to a standard Java Properties File. The entry is read and processed to obtain the properties and their values.

See Also The properties element of a Component Description.

112.13.4.9 String xmlns default ""

□ The XML name space of the Component Description for this Component.

If not specified, the XML name space of the Component Description for this Component should be the lowest Declarative Services XML name space which supports all the specification features used by this Component.

See Also The XML name space specified for a Component Description.

112.13.4.10 ConfigurationPolicy configurationPolicy default OPTIONAL

□ The configuration policy of this Component.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID equals the name of the component.

If not specified, the configuration policy is based upon whether the component is also annotated with the Meta Type Designate annotation.

- Not annotated with `Designate` - The configuration policy is OPTIONAL.
- Annotated with `Designate(factory=false)` - The configuration policy is OPTIONAL.
- Annotated with `Designate(factory=true)` - The configuration policy is REQUIRE.

See Also The configuration-policy attribute of the component element of a Component Description.

Since 1.1

112.13.4.11 String[] configurationPid default "$"

□ The configuration PIDs for the configuration of this Component.

Each value specifies a configuration PID for this Component.

If no value is specified, the name of this Component is used as the configuration PID of this Component.

A special string ("`$`") can be used to specify the name of the component as a configuration PID. The NAME constant holds this special string. For example:

```
@Component(configurationPid={"com.acme.system", Component.NAME})
```

Tools creating a Component Description from this annotation must replace the special string with the actual name of this Component.

See Also The configuration-pid attribute of the component element of a Component Description.

Since 1.2

112.13.4.12 ServiceScope scope default DEFAULT

□ The service scope for the service of this Component.

If not specified (and the deprecated servicefactory() element is not specified), the singleton service scope is used. If the factory() element is specified or the immediate() element is specified with `true`, this element can only be specified with the singleton service scope.

See Also The scope attribute of the service element of a Component Description.

Since 1.3

112.13.4.13 Reference[] reference default {}

□ The lookup strategy references of this Component.

To access references using the lookup strategy, Reference annotations are specified naming the reference and declaring the type of the referenced service. The referenced service can be accessed using one of the `locateService` methods of `ComponentContext`.

To access references using method injection, bind methods are annotated with Reference. To access references using field injection, fields are annotated with Reference. To access references using constructor injection, constructor parameters are annotated with Reference.

See Also The reference element of a Component Description.

Since 1.3

112.13.4.14 String[] factoryProperty default {}

□ Factory properties for this Factory Component.

Each factory property string is specified as "name=value". The type of the factory property value can be specified in the name as name:type=value. The type must be one of the factory property types supported by the type attribute of the factory-property element of a Component Description.

To specify a factory property with multiple values, use multiple name, value pairs. For example, {"foo=bar", "foo=baz"}.

If specified, the factory() element must also be specified to indicate the component is a Factory Component.

See Also The factory-property element of a Component Description.

Since 1.4

112.13.4.15 String[] factoryProperties default {}

□ Factory property entries for this Factory Component.

Specifies the name of an entry in the bundle whose contents conform to a standard Java Properties File. The entry is read and processed to obtain the factory properties and their values.

If specified, the factory() element must also be specified to indicate the component is a Factory Component.

See Also The factory-properties element of a Component Description.

Since 1.4

112.13.4.16 String NAME = "$"

Special string representing the name of this Component.

This string can be used in configurationPid() to specify the name of the component as a configuration PID. For example:

```
@Component(configurationPid={"com.acme.system", Component.NAME})
```

Tools creating a Component Description from this annotation must replace the special string with the actual name of this Component.

Since 1.3

112.13.5 @ComponentPropertyType

Identify the annotated annotation as a Component Property Type.

Component Property Types can be applied as annotations to the implementation class of the Component. They can also be used as activation objects which means they can be used as parameter types for the component's constructor and life cycle methods Activate, Deactivate, and Modified as well as activation fields.

Component Property Types do not have to be annotated with this annotation to be used as parameter types but they must be annotated with this annotation to be used as annotations on the implementation class of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also Component Property Types.

Since 1.4

Retention CLASS

Target ANNOTATION_TYPE

112.13.6 enum ConfigurationPolicy

Configuration Policy for the Component annotation.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component.

Since 1.1

112.13.6.1 OPTIONAL

Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present.

112.13.6.2 REQUIRE

There must be a corresponding Configuration object for the component configuration to become satisfied.

112.13.6.3 IGNORE

Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.

112.13.6.4 public String toString()

112.13.6.5 public static ConfigurationPolicy valueOf(String name)

112.13.6.6 public static ConfigurationPolicy[] values()

112.13.7 @Deactivate

Identify the annotated method as the `deactivate` method of a Service Component.

The annotated method is the deactivate method of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also The deactivate attribute of the component element of a Component Description.

Since 1.1

Retention CLASS

Target METHOD

112.13.8 enum FieldOption

Field options for the Reference annotation.

Since 1.3

112.13.8.1 UPDATE

The update field option is used to update the collection referenced by the field when there are changes to the bound services.

This field option can only be used when the field reference has dynamic policy and multiple cardinality.

112.13.8.2 REPLACE

The replace field option is used to replace the field value with a new value when there are changes to the bound services.

112.13.8.3 public String toString()

112.13.8.4 public static FieldOption valueOf(String name)

112.13.8.5 public static FieldOption[] values()

112.13.9 @Modified

Identify the annotated method as the modified method of a Service Component.

The annotated method is the modified method of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also The modified attribute of the component element of a Component Description.

Since 1.1

Retention CLASS

Target METHOD

112.13.10 @Reference

Identify the annotated member or parameter as a reference of a Service Component.

When the annotation is applied to a method, the method is the bind method of the reference.

When the annotation is applied to a field, the field will contain the bound service(s) of the reference.

When the annotation is applied to a parameter of a constructor, the parameter will contain the bound service(s) of the reference.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

In the generated Component Description for a component, the references must be ordered in ascending lexicographical order (using String.compareTo) of the reference names.

See Also The reference element of a Component Description.

Retention CLASS

Target METHOD, FIELD, PARAMETER

112.13.10.1 String name default ""

□ The name of this reference.

The name of this reference must be specified when using this annotation in the Component.reference() element since there is no annotated member from which the name can be determined. If not specified, the name of this reference is based upon how this annotation is used:

- Annotated method - If the method name begins with bind, set or add, that prefix is removed to create the name of the reference. Otherwise, the name of the reference is the method name.
- Annotated field - The name of the reference is the field name.
- Annotated constructor parameter - The name of the reference is the parameter name.

See Also The name attribute of the reference element of a Component Description.

112.13.10.2 Class<?> service default Object.class

□ The type of the service for this reference.

The type of the service for this reference must be specified when using this annotation in the Component.reference() element since there is no annotated member from which the type of the service can be determined.

If not specified, the type of the service for this reference is based upon how this annotation is used:

- Annotated method - The type of the service is the type of the first parameter of the method.
- Annotated field - The type of the service is based upon the cardinality of the reference and the type of the field being annotated. If the cardinality is either 0..n, or 1..n, the type of the field must be one of java.util.Collection, java.util.List, or a subtype of java.util.Collection, so the type of the service is the generic type of the collection. If the cardinality is either 0..1 or 1..1, and the type of the field is java.util.Optional, the type of the service is the generic type of the java.util.Optional. Otherwise, the type of the service is the type of the field.
- Annotated constructor parameter - The type of the service is based upon the cardinality of the reference and the type of the parameter being annotated. If the cardinality is either 0..n, or 1..n, the type of the parameter must be one of java.util.Collection or java.util.List, so the type of the service is the generic type of the collection. If the cardinality is either 0..1 or 1..1, and the type of the parameter is java.util.Optional, the type of the service is the generic type of the java.util.Optional. Otherwise, the type of the service is the type of the parameter.

See Also The interface attribute of the reference element of a Component Description., AnyService

112.13.10.3 ReferenceCardinality cardinality default MANDATORY

□ The cardinality of this reference.

If not specified, the cardinality of this reference is based upon how this annotation is used:

- Annotated method - The cardinality is 1..1.
- Annotated field - The cardinality is based on the type of the field. If the type is either java.util.Collection, java.util.List, or a subtype of java.util.Collection, the cardinality is 0..n. If the type is java.util.Optional, the cardinality is 0..1. Otherwise the cardinality is 1..1.
- Annotated constructor parameter - The cardinality is based on the type of the parameter. If the type is either java.util.Collection or java.util.List, the cardinality is 0..n. If the type is java.util.Optional, the cardinality is 0..1. Otherwise the cardinality is 1..1.
- Component.reference() element - The cardinality is 1..1.

See Also The cardinality attribute of the reference element of a Component Description.

112.13.10.4 ReferencePolicy policy default STATIC

□ The policy for this reference.

If not specified, the policy of this reference is based upon how this annotation is used:

- Annotated method - The policy is STATIC.
- Annotated field - The policy is based on the modifiers of the field. If the field is declared volatile, the policy is ReferencePolicy.DYNAMIC. Otherwise the policy is STATIC.
- Annotated constructor parameter - The policy is STATIC. STATIC policy must be used for constructor parameters.
- Component.reference() element - The policy is STATIC.

See Also The policy attribute of the reference element of a Component Description.

112.13.10.5 String target default ""

□ The target property for this reference.

If not specified, no target property is set. A target property must be specified if the service() element refers to AnyService.

See Also The target attribute of the reference element of a Component Description.

112.13.10.6 ReferencePolicyOption policyOption default RELUCTANT

□ The policy option for this reference.

If not specified, the RELUCTANT reference policy option is used.

See Also The policy-option attribute of the reference element of a Component Description.

Since 1.2

112.13.10.7 ReferenceScope scope default BUNDLE

□ The reference scope for this reference.

If not specified, the bundle reference scope is used.

See Also The scope attribute of the reference element of a Component Description.

Since 1.3

112.13.10.8 String bind default ""

□ The name of the bind method for this reference.

If specified and this reference annotates a method, the specified name must match the name of the annotated method.

If not specified, the name of the bind method is based upon how this annotation is used:

- Annotated method - The name of the annotated method is the name of the bind method.
- Annotated field - There is no bind method name.
- Annotated constructor parameter - There is no bind method name.
- Component.reference() element - There is no bind method name.

If there is a bind method name, the component must contain a method with that name.

See Also The bind attribute of the reference element of a Component Description.

Since 1.3

112.13.10.9 String updated default ""

□ The name of the updated method for this reference.

If not specified, the name of the updated method is based upon how this annotation is used:

- Annotated method - The name of the updated method is created from the name of the annotated method. If the name of the annotated method begins with bind, set or add, that prefix is replaced with updated to create the name candidate for the updated method. Otherwise, updated is prefixed to the name of the annotated method to create the name candidate for the updated method. If the component type contains a method with the candidate name, the candidate name is used as the name of the updated method. To declare no updated method when the component type contains a method with the candidate name, the value "-" must be used.
- Annotated field - There is no updated method name.
- Annotated constructor parameter - There is no updated method name.
- Component.reference() element - There is no updated method name.

If there is an updated method name, the component must contain a method with that name.

See Also The updated attribute of the reference element of a Component Description.

Since 1.2

112.13.10.10 String unbind default ""

□ The name of the unbind method for this reference.

If not specified, the name of the unbind method is based upon how this annotation is used:

- Annotated method - The name of the unbind method is created from the name of the annotated method. If the name of the annotated method begins with bind, set or add, that prefix is replaced with unbind, unset or remove, respectively, to create the name candidate for the unbind method. Otherwise, un is prefixed to the name of the annotated method to create the name candidate for the unbind method. If the component type contains a method with the candidate name, the candidate name is used as the name of the unbind method. To declare no unbind method when the component type contains a method with the candidate name, the value "-" must be used.
- Annotated field - There is no unbind method name.
- Annotated constructor parameter - There is no unbind method name.
- Component.reference() element - There is no unbind method name.

If there is an unbind method name, the component must contain a method with that name.

See Also The unbind attribute of the reference element of a Component Description.

112.13.10.11 String field default ""

□ The name of the field for this reference.

If specified and this reference annotates a field, the specified name must match the name of the annotated field.

If not specified, the name of the field is based upon how this annotation is used:

- Annotated method - There is no field name.
- Annotated field - The name of the annotated field is the name of the field.
- Annotated constructor parameter - There is no field name.
- Component.reference() element - There is no field name.

If there is a field name, the component must contain a field with that name.

See Also The field attribute of the reference element of a Component Description.

Since 1.3

112.13.10.12 FieldOption fieldOption default REPLACE

□ The field option for this reference.

If not specified, the field option is based upon how this annotation is used:

- Annotated method - There is no field option.
- Annotated field - The field option is based upon the policy and cardinality of the reference and the modifiers of the field. If the policy is ReferencePolicy.DYNAMIC, the cardinality is 0..n or 1..n, and the field is declared final, the field option is FieldOption.UPDATE. Otherwise, the field option is FieldOption.REPLACE.
- Annotated constructor parameter - There is no field option.
- Component.reference() element - There is no field option.

See Also The field-option attribute of the reference element of a Component Description.

Since 1.3

112.13.10.13 int parameter default 0

□ The zero-based parameter number of the constructor parameter for this reference.

If specified and this reference annotates an constructor parameter, the specified value must match the zero-based parameter number of the annotated constructor parameter.

If not specified, the parameter number is based upon how this annotation is used:

- Annotated method - There is no parameter number.
- Annotated field - There is no parameter number.
- Annotated constructor parameter - The zero-based parameter number of the parameter.
- Component.reference() element - There is no parameter number.

If there is a parameter number, the component must declare a constructor that has a parameter having the zero-based parameter number.

See Also The parameter attribute of the reference element of a Component Description., The init attribute of the component element of a Component Description.

Since 1.4

112.13.10.14 CollectionType collectionType default SERVICE

□ The collection type for this reference.

If not specified, the collection type is based upon how this annotation is used:

- Annotated method - There is no collection type.
- Annotated field - The collection type is based upon the cardinality of the reference and the type of the field. If the cardinality is either 0..n or 1..n, the collection type is inferred from the generic type of the java.util.Collection. If the cardinality is either 0..1 or 1..1, and the type of the field is java.util.Optional, the collection type is inferred from the generic type of the java.util.Optional. Otherwise, there is no collection type
- Annotated constructor method parameter - The collection type is based upon the cardinality of the reference and the type of the parameter. If the cardinality is either 0..n or 1..n, the collection type is inferred from the generic type of the java.util.Collection. If the cardinality is either 0..1 or 1..1, and the type of the parameter is java.util.Optional, the collection type is inferred from the generic type of the java.util.Optional. Otherwise, there is no collection type
- Component.reference() element - There is no collection type.

See Also The field-collection-type attribute of the reference element of a Component Description.

Since 1.4

112.13.11 enum ReferenceCardinality

Cardinality for the Reference annotation.

Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services.

112.13.11.1 OPTIONAL

The reference is optional and unary. That is, the reference has a cardinality of 0..1.

112.13.11.2 MANDATORY

The reference is mandatory and unary. That is, the reference has a cardinality of 1..1.

112.13.11.3 MULTIPLE

The reference is optional and multiple. That is, the reference has a cardinality of `0..n`.

112.13.11.4 AT_LEAST_ONE

The reference is mandatory and multiple. That is, the reference has a cardinality of `1..n`.

112.13.11.5 public String toString()

112.13.11.6 public static ReferenceCardinality valueOf(String name)

112.13.11.7 public static ReferenceCardinality[] values()

112.13.12 enum ReferencePolicy

Policy for the Reference annotation.

112.13.12.1 STATIC

The static policy is the most simple policy and is the default policy. A component instance never sees any of the dynamics. Component configurations are deactivated before any bound service for a reference having a static policy becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and bound to the replacement service.

112.13.12.2 DYNAMIC

The dynamic policy is slightly more complex since the component implementation must properly handle changes in the set of bound services. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses method injection to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind and unbind methods.

112.13.12.3 public String toString()

112.13.12.4 public static ReferencePolicy valueOf(String name)

112.13.12.5 public static ReferencePolicy[] values()

112.13.13 enum ReferencePolicyOption

Policy option for the Reference annotation.

Since 1.2

112.13.13.1 RELUCTANT

The reluctant policy option is the default policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the reluctant policy option for the static policy or the dynamic policy with a unary cardinality will ignore the new target service. References having the dynamic policy with a multiple cardinality will bind the new target service.

112.13.13.2 GREEDY

The greedy policy option is a valid policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the greedy policy option will bind the new target service.

112.13.13.3 public String toString()

112.13.13.4 public static ReferencePolicyOption valueOf(String name)

112.13.13.5 public static ReferencePolicyOption[] values()

112.13.14 enum ReferenceScope

Reference scope for the Reference annotation.

Since 1.3

112.13.14.1 BUNDLE

A single service object is used for all references to the service in this bundle.

112.13.14.2 PROTOTYPE

If the bound service has prototype service scope, then each instance of the component with this reference can receive a unique instance of the service. If the bound service does not have prototype service scope, then this reference scope behaves the same as BUNDLE.

112.13.14.3 PROTOTYPE_REQUIRED

Bound services must have prototype service scope. Each instance of the component with this reference can receive a unique instance of the service.

112.13.14.4 public String toString()

112.13.14.5 public static ReferenceScope valueOf(String name)

112.13.14.6 public static ReferenceScope[] values()

112.13.15 @RequireServiceComponentRuntime

This annotation can be used to require the Service Component Runtime to process Declarative Services components. It can be used directly, or as a meta-annotation.

Since 1.4

Retention CLASS

Target TYPE, PACKAGE

112.13.16 enum ServiceScope

Service scope for the Component annotation.

Since 1.3

112.13.16.1 SINGLETON

When the component is registered as a service, it must be registered as a bundle scope service but only a single instance of the component must be used for all bundles using the service.

112.13.16.2 BUNDLE

When the component is registered as a service, it must be registered as a bundle scope service and an instance of the component must be created for each bundle using the service.

112.13.16.3 PROTOTYPE

When the component is registered as a service, it must be registered as a prototype scope service and an instance of the component must be created for each distinct request for the service.

112.13.16.4 DEFAULT

Default element value for annotation. This is used to distinguish the default value for an element and should not otherwise be used.

112.13.16.5 public String toString()

112.13.16.6 public static ServiceScope valueOf(String name)

112.13.16.7 public static ServiceScope[] values()

112.14 org.osgi.service.component.runtime

Service Component Runtime Package Version 1.5.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.component.runtime; version="[1.5,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.component.runtime; version="[1.5,1.6)"`

112.14.1 Summary

- `ServiceComponentRuntime` - The `ServiceComponentRuntime` service represents the Declarative Services actor, known as Service Component Runtime (SCR), that manages the service components and their life cycle.

112.14.2 public interface ServiceComponentRuntime

The `ServiceComponentRuntime` service represents the Declarative Services actor, known as Service Component Runtime (SCR), that manages the service components and their life cycle. The `ServiceComponentRuntime` service allows introspection of the components managed by Service Component Runtime.

This service differentiates between a ComponentDescriptionDTO and a ComponentConfigurationDTO. A ComponentDescriptionDTO is a representation of a declared component description. A ComponentConfigurationDTO is a representation of an actual instance of a declared component description parameterized by component properties.

This service must be registered with a Constants.SERVICE_CHANGECOUNT service property that must be updated each time the SCR DTOs available from this service change.

Access to this service requires the `ServicePermission[ServiceComponentRuntime, GET]` permission. It is intended that only administrative bundles should be granted this permission to limit access to the potentially intrusive methods provided by this service.

Since 1.3

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.14.2.1 public Promise<Void> disableComponent(ComponentDescriptionDTO description)

description The component description to disable. Must not be null.

□ Disables the specified component description.

If the specified component description is currently disabled, this method has no effect.

This method must return after changing the enabled state of the specified component description. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Returns A promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed. If the provided description does not belong to an active bundle, a failed promise is returned.

See Also isComponentEnabled(ComponentDescriptionDTO)

112.14.2.2 public Promise<Void> enableComponent(ComponentDescriptionDTO description)

description The component description to enable. Must not be null.

□ Enables the specified component description.

If the specified component description is currently enabled, this method has no effect.

This method must return after changing the enabled state of the specified component description. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Returns A promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed. If the provided description does not belong to an active bundle, a failed promise is returned.

See Also isComponentEnabled(ComponentDescriptionDTO)

112.14.2.3 public Collection<ComponentConfigurationDTO> getComponentConfigurationDTOs(ComponentDescriptionDTO description)

description The component description. Must not be null.

□ Returns the component configurations for the specified component description.

Returns A collection containing a snapshot of the current component configurations for the specified component description. An empty collection is returned if there are none or if the provided component description does not belong to an active bundle.

112.14.2.4 public ComponentDescriptionDTO getComponentDescriptionDTO(Bundle bundle, String name)

bundle The bundle declaring the component description. Must not be null.

name The name of the component description. Must not be null.

□ Returns the ComponentDescriptionDTO declared with the specified name by the specified bundle.

Only component descriptions from active bundles are returned. null if no such component is declared by the given bundle or the bundle is not active.

Returns The declared component description or null if the specified bundle is not active or does not declare a component description with the specified name.

112.14.2.5 public Collection<ComponentDescriptionDTO> getComponentDescriptionDTOs(Bundle... bundles)

bundles The bundles whose declared component descriptions are to be returned. Specifying no bundles, or the equivalent of an empty Bundle array, will return the declared component descriptions from all active bundles.

□ Returns the component descriptions declared by the specified active bundles.

Only component descriptions from active bundles are returned. If the specified bundles have no declared components or are not active, an empty collection is returned.

Returns The declared component descriptions of the specified active bundles. An empty collection is returned if there are no component descriptions for the specified active bundles.

112.14.2.6 public boolean isComponentEnabled(ComponentDescriptionDTO description)

description The component description. Must not be null.

□ Returns whether the specified component description is currently enabled.

The enabled state of a component description is initially set by the enabled attribute of the component description.

Returns true if the specified component description is currently enabled. Otherwise, false.

See Also enableComponent(ComponentDescriptionDTO), disableComponent(ComponentDescriptionDTO), ComponentContext.disableComponent(String), ComponentContext.enableComponent(String)

112.15 org.osgi.service.component.runtime.dto

Service Component Runtime Data Transfer Objects Package Version 1.5.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.component.runtime.dto; version="[1.5,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.component.runtime.dto; version="[1.5,1.6)"
```

112.15.1 Summary

- ComponentConfigurationDTO - A representation of an actual instance of a declared component description parameterized by component properties.
- ComponentDescriptionDTO - A representation of a declared component description.
- ReferenceDTO - A representation of a declared reference to a service.
- SatisfiedReferenceDTO - A representation of a satisfied reference.
- UnsatisfiedReferenceDTO - A representation of an unsatisfied reference.

112.15.2 public class ComponentConfigurationDTO extends DTO

A representation of an actual instance of a declared component description parameterized by component properties.

Since 1.3

Concurrency Not Thread-safe

112.15.2.1 public static final int ACTIVE = 8

The component configuration is active.

This is the normal operational state of a component configuration.

112.15.2.2 public ComponentDescriptionDTO description

The representation of the component configuration's component description.

112.15.2.3 public static final int FAILED_ACTIVATION = 16

The component configuration failed to activate.

This means the component configuration is satisfied but that either:

- an exception occurred loading the implementation class,
- the static initializer threw an exception,
- the constructor threw an exception, or
- the activate method threw an exception.

The failure information from the exception is available from failure.

Since 1.4

112.15.2.4 public String failure

The failure information if the component configuration state is FAILED_ACTIVATION.

This is the failure exception converted to a String using:

```
StringWriter sw = new StringWriter();
exception.printStackTrace(new PrintWriter(sw));
sw.toString();
```

This must be null if the component configuration state is not FAILED_ACTIVATION.

Since 1.4

112.15.2.5 public long id

The id of the component configuration.

The id is a non-persistent, unique value assigned at runtime. The id is also available as the component.id component property. The value of this field is unspecified if the state of this component configuration is unsatisfied.

112.15.2.6 public Map<String, Object> properties

The component properties for the component configuration.

See Also ComponentContext.getProperties()

112.15.2.7 public static final int SATISFIED = 4

The component configuration is satisfied.

Any services declared by the component description are registered.

112.15.2.8 public SatisfiedReferenceDTO[] satisfiedReferences

The satisfied references.

Each SatisfiedReferenceDTO in the array represents a satisfied reference of the component configuration. The array must be empty if the component configuration has no satisfied references.

112.15.2.9 public ServiceReferenceDTO service

The registered service of the component configuration.

This must be non-null if the component configuration is registered as a service. Otherwise it must be null.

Since 1.4

112.15.2.10 public int state

The current state of the component configuration.

This is one of UNSATISFIED_CONFIGURATION, UNSATISFIED_REFERENCE, SATISFIED, ACTIVE, or FAILED_ACTIVATION.

112.15.2.11 public static final int UNSATISFIED_CONFIGURATION = 1

The component configuration is unsatisfied due to a missing required configuration.

112.15.2.12 public static final int UNSATISFIED_REFERENCE = 2

The component configuration is unsatisfied due to an unsatisfied reference.

112.15.2.13 public UnsatisfiedReferenceDTO[] unsatisfiedReferences

The unsatisfied references.

Each UnsatisfiedReferenceDTO in the array represents an unsatisfied reference of the component configuration. The array must be empty if the component configuration has no unsatisfied references.

112.15.2.14 public ComponentConfigurationDTO()

112.15.3 public class ComponentDescriptionDTO extends DTO

A representation of a declared component description.

Since 1.3

Concurrency Not Thread-safe

112.15.3.1 public String activate

The name of the activate method.

This is declared in the activate attribute of the component element. This must be null if the component description does not declare an activate method name.

112.15.3.2 public String[] activationFields

The activation fields.

These are declared in the activation-fields attribute of the component element. The array must be empty if the component description does not declare any activation fields.

Since 1.4

112.15.3.3 public BundleDTO bundle

The bundle declaring the component description.

112.15.3.4 public String[] configurationPid

The configuration pids.

These are declared in the configuration-pid attribute of the component element. This must contain the default configuration pid if the component description does not declare a configuration pid.

112.15.3.5 public String configurationPolicy

The configuration policy.

This is declared in the configuration-policy attribute of the component element. This must be the default configuration policy if the component description does not declare a configuration policy.

112.15.3.6 **public String deactivate**

The name of the deactivate method.

This is declared in the deactivate attribute of the component element. This must be null if the component description does not declare a deactivate method name.

112.15.3.7 **public boolean defaultEnabled**

The initial enabled state.

This is declared in the enabled attribute of the component element.

112.15.3.8 **public String factory**

The component factory name.

This is declared in the factory attribute of the component element. This must be null if the component description is not declared as a factory component.

112.15.3.9 **public Map<String, Object> factoryProperties**

The factory properties.

These are declared in the component description by the factory-property and factory-properties elements. This must be null if the component description is not declared as a factory component.

Since 1.4

112.15.3.10 **public boolean immediate**

The immediate state.

This is declared in the immediate attribute of the component element.

112.15.3.11 **public String implementationClass**

The fully qualified name of the implementation class.

This is declared in the class attribute of the implementation element.

112.15.3.12 **public int init**

The constructor parameter count.

This is declared in the init attribute of the component element. This must be 0 if the component description does not declare an init attribute.

Since 1.4

112.15.3.13 **public String modified**

The name of the modified method.

This is declared in the modified attribute of the component element. This must be null if the component description does not declare a modified method name.

112.15.3.14 **public String name**

The name of the component.

This is declared in the name attribute of the component element. This must be the default name if the component description does not declare a name.

112.15.3.15 **public Map<String, Object> properties**

The component properties.

These are declared in the component description by the property and properties elements as well as the target attribute of the reference elements.

112.15.3.16 public ReferenceDTO[] references

The referenced services.

These are declared in the reference elements. The array must be empty if the component description does not declare references to any services.

112.15.3.17 public String scope

The service scope.

This is declared in the scope attribute of the service element. This must be null if the component description does not declare any service interfaces.

112.15.3.18 public String[] serviceInterfaces

The fully qualified names of the service interfaces.

These are declared in the interface attribute of the provide elements. The array must be empty if the component description does not declare any service interfaces.

112.15.3.19 public ComponentDescriptionDTO()

112.15.4 public class ReferenceDTO extends DTO

A representation of a declared reference to a service.

Since 1.3

Concurrency Not Thread-safe

112.15.4.1 public String bind

The name of the bind method of the reference.

This is declared in the bind attribute of the reference element. This must be null if the component description does not declare a bind method for the reference.

112.15.4.2 public String cardinality

The cardinality of the reference.

This is declared in the cardinality attribute of the reference element. This must be the default cardinality if the component description does not declare a cardinality for the reference.

112.15.4.3 public String collectionType

The collection type for the reference.

This is declared in the field-collection-type attribute of the reference element. This must be null if the component description does not declare a collection type for the reference.

Since 1.4

112.15.4.4 public String field

The name of the field of the reference.

This is declared in the field attribute of the reference element. This must be null if the component description does not declare a field for the reference.

112.15.4.5 public String fieldOption

The field option of the reference.

This is declared in the field-option attribute of the reference element. This must be null if the component description does not declare a field for the reference.

112.15.4.6 public String interfaceName

The service interface of the reference.

This is declared in the interface attribute of the reference element.

112.15.4.7 public String name

The name of the reference.

This is declared in the name attribute of the reference element. This must be the default name if the component description does not declare a name for the reference.

112.15.4.8 public Integer parameter

The zero-based parameter number of the constructor parameter for the reference.

This is declared in the parameter attribute of the reference element. This must be null if the component description does not declare a parameter number for the reference.

Since 1.4

112.15.4.9 public String policy

The policy of the reference.

This is declared in the policy attribute of the reference element. This must be the default policy if the component description does not declare a policy for the reference.

112.15.4.10 public String policyOption

The policy option of the reference.

This is declared in the policy-option attribute of the reference element. This must be the default policy option if the component description does not declare a policy option for the reference.

112.15.4.11 public String scope

The scope of the reference.

This is declared in the scope attribute of the reference element. This must be the default scope if the component description does not declare a scope for the reference.

112.15.4.12 public String target

The target of the reference.

This is declared in the target attribute of the reference element. This must be null if the component description does not declare a target for the reference.

112.15.4.13 public String unbind

The name of the unbind method of the reference.

This is declared in the unbind attribute of the reference element. This must be null if the component description does not declare an unbind method for the reference.

112.15.4.14 public String updated

The name of the updated method of the reference.

This is declared in the updated attribute of the reference element. This must be null if the component description does not declare an updated method for the reference.

112.15.4.15 public ReferenceDTO()

112.15.5 public class SatisfiedReferenceDTO extends DTO

A representation of a satisfied reference.

Since 1.3

Concurrency Not Thread-safe

112.15.5.1 public ServiceReferenceDTO[] boundServices

The bound services.

Each ServiceReferenceDTO in the array represents a service bound to the satisfied reference. The array must be empty if there are no bound services.

112.15.5.2 public String name

The name of the declared reference.

This is declared in the `name` attribute of the `reference` element of the component description.

See Also ReferenceDTO.name

112.15.5.3 public String target

The target property of the satisfied reference.

This is the value of the component property whose name is the concatenation of the declared reference name and ".target". This must be `null` if no target property is set for the reference.

112.15.5.4 public SatisfiedReferenceDTO()

112.15.6 public class UnsatisfiedReferenceDTO extends DTO

A representation of an unsatisfied reference.

Since 1.3

Concurrency Not Thread-safe

112.15.6.1 public String name

The name of the declared reference.

This is declared in the `name` attribute of the `reference` element of the component description.

See Also ReferenceDTO.name

112.15.6.2 public String target

The target property of the unsatisfied reference.

This is the value of the component property whose name is the concatenation of the declared reference name and ".target". This must be `null` if no target property is set for the reference.

112.15.6.3 public ServiceReferenceDTO[] targetServices

The target services.

Each ServiceReferenceDTO in the array represents a target service for the reference. The array must be empty if there are no target services. The upper bound on the number of target services in the array is the upper bound on the cardinality of the reference.

112.15.6.4 public UnsatisfiedReferenceDTO()

112.16 org.osgi.service.component.propertytypes

Component Property Types Package Version 1.5.

When used as annotations, component property types are processed by tools to generate Component Descriptions which are used at runtime.

Bundles wishing to use this package at runtime must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.component.propertytypes; version="[1.5,2.0)"
```

112.16.1 Summary

- ExportedService - Component Property Type for the remote service properties for an exported service.
- SatisfyingConditionTarget - Component Property Type for the osgi.ds.satisfying.condition.target reference property.
- ServiceDescription - Component Property Type for the service.description service property.
- ServiceRanking - Component Property Type for the service.ranking service property.
- ServiceVendor - Component Property Type for the service.vendor service property.

112.16.2 @ExportedService

Component Property Type for the remote service properties for an exported service.

This annotation can be used on a Component to declare the values of the remote service properties for an exported service.

See Also Component Property Types, Remote Services Specification

Since 1.4

Retention CLASS

Target TYPE

112.16.2.1 Class<?>[] service_exported_interfaces

□ Service property marking the service for export. It defines the interfaces under which the service can be exported.

If an empty array is specified, the property is not added to the component description.

Returns The exported service interfaces.

See Also Constants.SERVICE_EXPORTED_INTERFACES

112.16.2.2 String[] service_exported_configs default {}

□ Service property identifying the configuration types that should be used to export the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The configuration types.

See Also Constants.SERVICE_EXPORTED_CONFIGS

112.16.2.3 String[] service_exported_intents default {}

□ Service property identifying the intents that the distribution provider must implement to distribute the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The intents that the distribution provider must implement to distribute the service.

See Also Constants.SERVICE_EXPORTED_INTENTS

112.16.2.4 String[] service_exported_intents_extra default {}

□ Service property identifying the extra intents that the distribution provider must implement to distribute the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The extra intents that the distribution provider must implement to distribute the service.

See Also Constants.SERVICE_EXPORTED_INTENTS_EXTRA

112.16.2.5 String[] service_intents default {}

□ Service property identifying the intents that this service implements.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The intents that the service implements.

See Also Constants.SERVICE_INTENTS

112.16.3 @SatisfyingConditionTarget

Component Property Type for the osgi.ds.satisfying.condition.target reference property.

This annotation can be used on a Component to declare the value of the target property for the component's satisfying condition reference if a value other than the default value is desired.

See Also Component Property Types

Since 1.5

Retention CLASS

Target TYPE

112.16.3.1 String value default "(osgi.condition.id=true)"

□ Filter expression to select the component's satisfying condition.

Returns The filter expression to select the component's satisfying condition.

112.16.3.2 String PREFIX_ = "osgi.ds."

Prefix for the property name. This value is prepended to each property name.

112.16.4 @ServiceDescription

Component Property Type for the service.description service property.

This annotation can be used on a Component to declare the value of the Constants.SERVICE_DESCRIPTION service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

112.16.4.1 String value

□ Service property identifying a service's description.

Returns The service description.

See Also Constants.SERVICE_DESCRIPTION

112.16.5 @ServiceRanking

Component Property Type for the service.ranking service property.

This annotation can be used on a Component to declare the value of the Constants.SERVICE_RANKING service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

112.16.5.1 int value

□ Service property identifying a service's ranking.

Returns The service ranking.

See Also Constants.SERVICE_RANKING

112.16.6 @ServiceVendor

Component Property Type for the service.vendor service property.

This annotation can be used on a Component to declare the value of the Constants.SERVICE_VENDOR service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

112.16.6.1 String value

□ Service property identifying a service's vendor.

Returns The service vendor.

See Also Constants.SERVICE_VENDOR

112.17 References

[1] *Automating Service Dependency Management in a Service-Oriented Component Model*
Humberto Cervantes, Richard S. Hall, Proceedings of the Sixth Component-Based Software Engineering Workshop, May 2003, pp. 91-96
https://www.researchgate.net/publication/238655490_Automating_Service_Dependency_Management_in_a_Service-Oriented_Component_Model

[2] *Service Binder*
Humberto Cervantes, Richard S. Hall
http://gravity.sourceforge.net/servicebinder

[3] *Java Properties File*
https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html

[4] *Extensible Markup Language (XML) 1.0*
https://www.w3.org/TR/REC-xml/

[5] *OSGi XML Schemas*
https://docs.osgi.org/xmlns/

[6] *The Java Virtual Machine Specification, Java SE 8 Edition*
https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[7] *The Java Language Specification, Java SE 8 Edition*
https://docs.oracle.com/javase/specs/jls/se8/html/index.html

113 Event Admin Service Specification

Version 1.4

113.1 Introduction

Nearly all the bundles in an OSGi framework must deal with events, either as an event publisher or as an event handler. So far, the preferred mechanism to disperse those events have been the service interface mechanism.

Dispatching events for a design related to X, usually involves a service of type XListener. However, this model does not scale well for fine grained events that must be dispatched to many different handlers. Additionally, the dynamic nature of the OSGi environment introduces several complexities because both event publishers and event handlers can appear and disappear at any time.

The Event Admin service provides an inter-bundle communication mechanism. It is based on a event *publish* and *subscribe* model, popular in many message based systems.

This specification defines the details for the participants in this event model.

113.1.1 Essentials

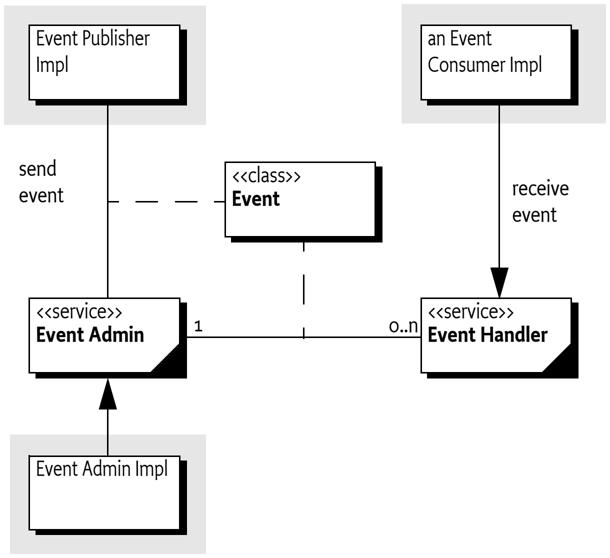
- *Simplifications* - The model must significantly simplify the process of programming an event source and an event handler.
- *Dependencies* - Handle the myriad of dependencies between event sources and event handlers for proper cleanup.
- *Synchronicity* - It must be possible to deliver events asynchronously or synchronously with the caller.
- *Event Window* - Only event handlers that are active when an event is published must receive this event, handlers that register later must not see the event.
- *Performance* - The event mechanism must impose minimal overhead in delivering events.
- *Selectivity* - Event listeners must only receive notifications for the event types for which they are interested
- *Reliability* - The Event Admin must ensure that events continue to be delivered regardless the quality of the event handlers.
- *Security* - Publishing and receiving events are sensitive operations that must be protected per event type.
- *Extensibility* - It must be possible to define new event types with their own data types.
- *Native Code* - Events must be able to be passed to native code or come from native code.
- *OSGi Events* - The OSGi Framework, as well as a number of OSGi services, already have number of its own events defined. For uniformity of processing, these have to be mapped into generic event types.

113.1.2 Entities

- *Event* - An Event object has a topic and a Dictionary object that contains the event properties. It is an immutable object.
- *Event Admin* - The service that provides the publish and subscribe model to Event Handlers and Event Publishers.

- *Event Handler* - A service that receives and handles `Event` objects.
- *Event Publisher* - A bundle that sends event through the Event Admin service.
- *Event Subscriber* - Another name for an Event Handler.
- *Topic* - The name of an Event type.
- *Event Properties* - The set of properties that is associated with an Event.

Figure 113.1 *The Event Admin service org.osgi.service.event package*



113.1.3 Synopsis

The Event Admin service provides a place for bundles to publish events, regardless of their destination. It is also used by Event Handlers to subscribe to specific types of events.

Events are published under a topic, together with a number of event properties. Event Handlers can specify a filter to control the Events they receive on a very fine grained basis.

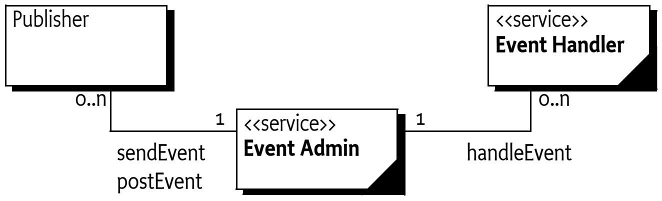
113.1.4 What To Read

- *Architects* - The *Event Admin Architecture* on page 342 provides an overview of the Event Admin service.
- *Event Publishers* - The *Event Publisher* on page 346 provides an introduction of how to write an Event Publisher. The *Event Admin Architecture* on page 342 provides a good overview of the design.
- *Event Subscribers/Handlers* - The *Event Handler* on page 344 provides the rules on how to subscribe and handle events.

113.2 Event Admin Architecture

The Event Admin is based on the *Publish-Subscribe* pattern. This pattern decouples sources from their handlers by interposing an *event channel* between them. The publisher posts events to the channel, which identifies which handlers need to be notified and then takes care of the notification process. This model is depicted in Figure 113.2.

Figure 113.2 *Channel Pattern*



In this model, the event source and event handler are completely decoupled because neither has any direct knowledge of the other. The complicated logic of monitoring changes in the event publishers and event handlers is completely contained within the event channel. This is highly advantageous in an OSGi environment because it simplifies the process of both sending and receiving events.

113.3 The Event

Events have the following attributes:

- *Topic* - A topic that defines what happened. For example, when a bundle is started an event is published that has a topic of org/osgi/framework/BundleEvent/STARTED.
- *Properties* - Zero or more properties that contain additional information about the event. For example, the previous example event has a property of bundle.id which is set to a Long object, among other properties.

113.3.1 Topics

The topic of an event defines the *type* of the event. It is fairly granular in order to give handlers the opportunity to register for just the events they are interested in. When a topic is designed, its name should not include any other information, such as the publisher of the event or the data associated with the event, those parts are intended to be stored in the event properties.

The topic is intended to serve as a first-level filter for determining which handlers should receive the event. Event Admin service implementations use the structure of the topic to optimize the dispatching of the events to the handlers.

Topics are arranged in a hierarchical namespace. Each level is defined by a token and levels are separated by solidi ('/' \u002F). More precisely, the topic must conform to the following grammar:

```
  topic ::= token ( '/' token ) *    // See General Syntax Definitions in Core
```

Topics should be designed to become more specific when going from left to right. Handlers can provide a prefix that matches a topic, using the preferred order allows a handler to minimize the number of prefixes it needs to register.

Topics are case-sensitive. As a convention, topics should follow the reverse domain name scheme used by Java packages to guarantee uniqueness. The separator must be a solidus ('/' \u002F) instead of the full stop ('.' \u002E).

This specification uses the convention fully/qualified/package/ClassName/ACTION. If necessary, a pseudo-class-name is used.

113.3.2 Properties

Information about the actual event is provided as properties. The property name is a case-sensitive string and the value can be any object. Although any Java object can be used as a property value, only String objects and the eight primitive types (plus their wrappers) should be used. Other types cannot be passed to handlers that reside external from the Java VM.

Another reason that arbitrary classes should not be used is the mutability of objects. If the values are not immutable, then any handler that receives the event could change the value. Any handlers that received the event subsequently would see the altered value and not the value as it was when the event was sent.

The topic of the event is available as a property with the key EVENT_TOPIC. This allows filters to include the topic as a condition if necessary.

113.3.3 High Performance

An event processing system can become a bottleneck in large systems. One expensive aspect of the Event object is its properties and its immutability. This combination requires the Event object to create a copy of the properties for each object. There are many situations where the same properties are dispatched through Event Admin, the topic is then used to signal the information. Creating the copy of the properties can therefore take unnecessary CPU time and memory. However, the immutability of the Event object requires the properties to be immutable.

For this reason, this specification also provides an immutable Map with the Event Properties class. This class implements an immutable map that is recognized and trusted by the Event object to not mutate. Using an Event Properties object allows a client to create many different Event objects with different topics but sharing the same properties object.

The following example shows how an event poster can limit the copying of the properties.

```
void foo(EventAdmin eventAdmin) {
   Map<String,Object> props = new HashMap<String,Object>();
   props.put("foo", 1);
   EventProperties eventProps = new EventProperties( props);

   for ( int i=0; i<1000; i++)
      eventAdmin.postEvent( new Event( "my/topic/" + i, eventProps));
}
```

113.4 Event Handler

Event handlers must be registered as services with the OSGi framework under the object class org.osgi.service.event.EventHandler.

Event handlers should be registered with a property (constant from the EventConstants class) EVENT_TOPIC. The value being a String, String[] or Collection<String> object that describes which *topics* the handler is interested in. A wildcard asterisk ('*' \u002A) may be used as the last token of a topic name, for example com/action/*. This matches any topic that shares the same first tokens. For example, com/action/* matches com/action/listen.

Event Handlers which have not specified the EVENT_TOPIC service property must not receive events.

The value of each entry in the EVENT_TOPIC service registration property must conform to the following grammar:

```
topic-scope ::= '*' | ( topic '/*'? )
```

The EventTopics component property type can be used for this property on Declarative Services components.

Event handlers can also be registered with a service property named EVENT_FILTER. The value of this property must be a string containing a Framework filter specification. Any of the event's properties can be used in the filter expression.

```
event-filter ::= filter              //  See Filter Syntax in Core
```

Each Event Handler is notified for any event which belongs to the topics the handler has expressed an interest in. If the handler has defined a EVENT_FILTER service property then the event properties must also match the filter expression. If the filter is an error, then the Event Admin service should log a warning and further ignore the Event Handler. The EventFilter component property type can be used for this property on Declarative Services components.

For example, a bundle wants to see all Log Service events with a level of WARNING or ERROR, but it must ignore the INFO and DEBUG events. Additionally, the only events of interest are when the bundle symbolic name starts with com.acme.

```
public AcmeWatchDog implements BundleActivator,
        EventHandler {
    final static String [] topics = new String[] {
        "org/osgi/service/log/LogEntry/LOG_WARNING",
        "org/osgi/service/log/LogEntry/LOG_ERROR" };

    public void start(BundleContext context) {
        Dictionary d = new Hashtable();
        d.put(EventConstants.EVENT_TOPIC, topics );
        d.put(EventConstants.EVENT_FILTER,
            "(bundle.symbolicName=com.acme.*)" );
        context.registerService( EventHandler.class.getName(),
            this, d );
    }
    public void stop( BundleContext context) {}

    public void handleEvent(Event event ) {
        //...
    }
}
```

If there are multiple Event Admin services registered with the Framework then all Event Admin services must send their published events to all registered Event Handlers.

113.4.1 Ordering

In the default case, an Event Handler will receive posted (asynchronous) events from a single thread in the same order as they were posted. Maintaining this ordering guarantee requires the Event Admin to serialize the delivery of events instead of, for example, delivering the events on different worker threads. There are many scenarios where this ordering is not really required. For this reason, an Event Handler can signal to the Event Admin that events can be delivered out of order. This is notified with the EVENT_DELIVERY service property. This service property can be used in the following way:

- Not set or set to both - The Event Admin must deliver the events in the proper order.
- DELIVERY_ASYNC_ORDERED - Events must be delivered in order.
- DELIVERY_ASYNC_UNORDERED - Allow the events to be delivered in any order.

The EventDelivery component property type can be used for this property on Declarative Services components.

113.5 Event Publisher

To fire an event, the event source must retrieve the Event Admin service from the OSGi service registry. Then it creates the event object and calls one of the Event Admin service's methods to fire the event either synchronously or asynchronously.

The following example is a class that publishes a time event every 60 seconds.

```
public class TimerEvent extends Thread
    implements BundleActivator {
    Hashtable      time = new Hashtable();
    ServiceTracker tracker;

    public TimerEvent() { super("TimerEvent"); }

    public void start(BundleContext context ) {
        tracker = new ServiceTracker(context,
            EventAdmin.class.getName(), null );
        tracker.open();
        start();
    }

    public void stop( BundleContext context ) {
        interrupt();
        tracker.close();
    }

    public void run() {
        while ( ! Thread.interrupted() ) try {
            Calendar c = Calendar.getInstance();
            set(c,Calendar.MINUTE,"minutes");
            set(c,Calendar.HOUR,"hours");
            set(c,Calendar.DAY_OF_MONTH,"day");
            set(c,Calendar.MONTH,"month");
            set(c,Calendar.YEAR,"year");

            EventAdmin ea =
                (EventAdmin) tracker.getService();
            if ( ea != null )
                ea.sendEvent(new Event("com/acme/timer",
                    time ));
            Thread.sleep(60000-c.get(Calendar.SECOND)*1000);
        } catch( InterruptedException e ) {
            return;
        }
    }

    void set( Calendar c, int field, String key ) {
        time.put( key, new Integer(c.get(field)) );
    }
}
```

113.6 Specific Events

113.6.1 General Conventions

Some handlers are more interested in the contents of an event rather than what actually happened. For example, a handler wants to be notified whenever an Exception is thrown anywhere in the system. Both Framework Events and Log Entry events may contain an exception that would be of interest to this hypothetical handler. If both Framework Events and Log Entries use the same property names then the handler can access the Exception in exactly the same way. If some future event type follows the same conventions then the handler can receive and process the new event type even though it had no knowledge of it when it was compiled.

The following properties are suggested as conventions. When new event types are defined they should use these names with the corresponding types and values where appropriate. These values should be set only if they are not null

A list of these property names can be found in the following table.

Table 113.1 *General property names for events*

Name	Type	Notes
BUNDLE_SIGNER	String | Collection <String>	A bundle's signers DN
BUNDLE_VERSION	Version	A bundle's version
BUNDLE_SYMBOLICNAME	String	A bundle's symbolic name
EVENT	Object	The actual event object. Used when rebroadcasting an event that was sent via some other event mechanism
EXCEPTION	Throwable	An exception or error
EXCEPTION_MESSAGE	String	Must be equal to exception.getMessage().
EXCEPTION_CLASS	String	Must be equal to the name of the Exception class.
MESSAGE	String	A human-readable message that is usually not localized.
SERVICE	Service Reference	A Service Reference
SERVICE_ID	Long	A service's id
SERVICE_OBJECTCLASS	String[]	A service's objectClass
SERVICE_PID	String | Collection <String>	A service's persistent identity. A PID that is specified with a String[] must be coerced into a Collection<String>.
TIMESTAMP	Long	The time when the event occurred, as reported by System.currentTimeMillis()

The topic of an OSGi event is constructed by taking the fully qualified name of the event class, substituting a solidus ('/' \u002F)for every full stop, and appending a solidus followed by the name of the constant that defines the event type. For example, the topic of

```
BundleEvent.STARTED
```

Event becomes

```
org/osgi/framework/BundleEvent/STARTED
```

If a type code for the event is unknown then the event must be ignored.

113.6.2 OSGi Events

In order to present a consistent view of all the events occurring in the system, the existing Framework-level events are mapped to the Event Admin's publish-subscribe model. This allows event subscribers to treat framework events exactly the same as other events.

It is the responsibility of the Event Admin service implementation to map these Framework events to its queue.

The properties associated with the event depends on its class as outlined in the following sections.

113.6.3 Framework Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/FrameworkEvent/<eventtype>
```

The following event types are supported:

```
STARTED
ERROR
PACKAGES_REFRESHED
STARTLEVEL_CHANGED
WARNING
INFO
```

Other events are ignored, no event will be send by the Event Admin. The following event properties must be set for a Framework Event.

- `event` - (`FrameworkEvent`) The original event object.

If the `FrameworkEvent` `getBundle` method returns a non-`null` value, the following fields must be set:

- `bundle.id` - (`Long`) The source's bundle id.
- `bundle.symbolicName` - (`String`) The source bundle's symbolic name. Only set if the bundle's symbolic name is not `null`.
- `bundle.version` - (`Version`) The version of the bundle, if set.
- `bundle.signer` - (`String|Collection<String>`) The DNs of the signers.
- `bundle` - (`Bundle`) The source bundle.

If the `FrameworkEvent` `getThrowable` method returns a non-`null` value:

- `exception.class` - (`String`) The fully-qualified class name of the attached Exception.
- `exception.message` -(`String`) The message of the attached exception. Only set if the Exception message is not `null`.
- `exception` - (Throwable) The Exception returned by the `getThrowable` method.

113.6.4 Bundle Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/BundleEvent/<event type>
```

The following event types are supported:

```
INSTALLED
STARTED
STOPPED
```

```
UPDATED
UNINSTALLED
RESOLVED
UNRESOLVED
```

Unknown events must be ignored.

The following event properties must be set for a Bundle Event. If listeners require synchronous delivery then they should register a Synchronous Bundle Listener with the Framework.

- event - (BundleEvent) The original event object.
- bundle.id - (Long) The source's bundle id.
- bundle.symbolicName - (String) The source bundle's symbolic name. Only set if the bundle's symbolic name is not null.
- bundle.version - (Version) The version of the bundle, if set.
- bundle.signer - (String|Collection<String>) The DNs of the signers.
- bundle - (Bundle) The source bundle.

113.6.5 Service Event

Service Events must be delivered asynchronously with the topic:

```
org/osgi/framework/ServiceEvent/<eventtype>
```

The following event types are supported:

```
REGISTERED
MODIFIED
UNREGISTERING
```

Unknown events must be ignored.

- event - (ServiceEvent) The original Service Event object.
- service - (ServiceReference) The result of the getServiceReference method
- service.id - (Long) The service's ID.
- service.pid - (String or Collection<String>) The service's persistent identity. Only set if not null. If the PID is specified as a String[] then it must be coerced into a Collection<String>.
- service.objectClass - (String[]) The service's object class.

113.6.6 Other Event Sources

Several OSGi service specifications define their own event model. It is the responsibility of these services to map their events to Event Admin events. Event Admin is seen as a core service that will be present in most devices. However, if there is no Event Admin service present, applications are not mandated to buffer events.

113.7 Event Admin Service

The Event Admin service must be registered as a service with the object class org.osgi.service.event.EventAdmin. Multiple Event Admin services can be registered. Publishers should publish their event on the Event Admin service with the highest value for the SERVICE_RANKING service property. This is the service selected by the getServiceReference method.

The Event Admin service is responsible for tracking the registered handlers, handling event notifications and providing at least one thread for asynchronous event delivery.

113.7.1 Synchronous Event Delivery

Synchronous event delivery is initiated by the `sendEvent` method. When this method is invoked, the Event Admin service determines which handlers must be notified of the event and then notifies each one in turn. The handlers can be notified in the caller's thread or in an event-delivery thread, depending on the implementation. In either case, all notifications must be completely handled before the `sendEvent` method returns to the caller.

Synchronous event delivery is significantly more expensive than asynchronous delivery. All things considered equal, the asynchronous delivery should be preferred over the synchronous delivery.

Callers of this method will need to be coded defensively and assume that synchronous event notifications could be handled in a separate thread. That entails that they must not be holding any monitors when they invoke the `sendEvent` method. Otherwise they significantly increase the likelihood of deadlocks because Java monitors are not reentrant from another thread by definition. Not holding monitors is good practice even when the event is dispatched in the same thread.

113.7.2 Asynchronous Event Delivery

Asynchronous event delivery is initiated by the `postEvent` method. When this method is invoked, the Event Admin service must determine which handlers are interested in the event. By collecting this list of handlers during the method invocation, the Event Admin service ensures that only handlers that were registered at the time the event was posted will receive the event notification. This is the same as described in *Delivering Events* of *OSGi Core Release 8*.

The Event Admin service can use more than one thread to deliver events. If it does then it must guarantee that each handler receives the events in the same order as the events were posted, unless this handler allows unordered deliver, see *Ordering* on page 345. This ensures that handlers see events in their expected order. For example, for some handlers it would be an error to see a `destroyed` event before the corresponding `created` event.

Before notifying each handler, the event delivery thread must ensure that the handler is still registered in the service registry. If it has been unregistered then the handler must not be notified.

113.7.3 Order of Event Delivery

Asynchronous events are delivered in the order in which they arrive in the event queue. Thus if two events are posted by the same thread then they will be delivered in the same order (though other events may come between them). However, if two or more events are posted by different threads then the order in which they arrive in the queue (and therefore the order in which they are delivered) will depend very much on subtle timing issues. The event delivery system cannot make any guarantees in this case. An Event Handler can indicate that the ordering is not relevant, allowing the Event Admin to more aggressively parallelize the event deliver, see *Ordering* on page 345.

Synchronous events are delivered as soon as they are sent. If two events are sent by the same thread, one after the other, then they must be guaranteed to be processed serially and in the same order. However, if two events are sent by different threads then no guarantees can be made. The events can be processed in parallel or serially, depending on whether or not the Event Admin service dispatches synchronous events in the caller's thread or in a separate thread.

Note that if the actions of a handler trigger a synchronous event, then the delivery of the first event will be paused and delivery of the second event will begin. Once delivery of the second event has completed, delivery of the first event will resume. Thus some handlers may observe the second event before they observe the first one.

113.8 Reliability

113.8.1 Exceptions in callbacks

If a handler throws an Exception during delivery of an event, it must be caught by the Event Admin service and handled in some implementation specific way. If a Log Service is available the exception should be logged. Once the exception has been caught and dealt with, the event delivery must continue with the next handlers to be notified, if any.

As the Log Service can also forward events through the Event Admin service there is a potential for a loop when an event is reported to the Log Service.

113.8.2 Dealing with Stalled Handlers

Event handlers should not spend too long in the handleEvent method. Doing so will prevent other handlers in the system from being notified. If a handler needs to do something that can take a while, it should do it in a different thread.

An event admin implementation can attempt to detect stalled or deadlocked handlers and deal with them appropriately. Exactly how it deals with this situation is left as implementation specific. One allowed implementation is to mark the current event delivery thread as invalid and spawn a new event delivery thread. Event delivery must resume with the next handler to be notified.

Implementations can choose to deny list any handlers that they determine are misbehaving. Deny listed handlers must not be notified of any events. If a handler is deny listed, the event admin should log a message that explains the reason for it.

113.9 Interoperability with Native Applications

Implementations of the Event Admin service can support passing events to, and/or receiving events from native applications.

If the implementation supports native interoperability, it must be able to pass the topic of the event and its properties to/from native code. Implementations must be able to support property values of the following types:

- String objects, including full Unicode support
- Integer, Long, Byte, Short, Float, Double, Boolean, Character objects
- Single-dimension arrays of the above types (including String)
- Single-dimension arrays of Java's eight primitive types (int, long, byte, short, float, double, boolean, char)

Implementations can support additional types. Property values of unsupported types must be silently discarded.

113.10 Capabilities

113.10.1 osgi.implementation Capability

The Event Admin implementation bundle must provide the osgi.implementation capability with the name EVENT_ADMIN_IMPLEMENTATION. This capability can be used by provisioning tools and during resolution to ensure that an Event Admin implementation is present. The capability must also declare a uses constraint for the org.osgi.service.event package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
       osgi.implementation="osgi.event";
       uses:="org.osgi.service.event";
       version:Version="1.4"
```

The RequireEventAdmin annotation can be used to require this capability.

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

113.10.2 osgi.service Capability

The bundle providing the Event Admin service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.event package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.event.EventAdmin";
  uses:="org.osgi.service.event"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

113.11 Security

113.11.1 Topic Permission

The TopicPermission class allows fine-grained control over which bundles may post events to a given topic and which bundles may receive those events.

The target parameter for the permission is the topic name. TopicPermission classes uses a wildcard matching algorithm similar to the BasicPermission class, except that solidi ('/' \u002F) are used as separators instead of full stop characters. For example, a name of a/b/* implies a/b/c but not x/y/z or a/b.

There are two available actions: PUBLISH and SUBSCRIBE. These control a bundle's ability to either publish or receive events, respectively. Neither one implies the other.

113.11.2 Required Permissions

Bundles that need to register an event handler must be granted ServicePermission[org.osgi.service.event.EventHandler, REGISTER]. In addition, handlers require TopicPermission[<topic>, SUBSCRIBE] for each topic they want to be notified about.

Bundles that need to publish an event must be granted ServicePermission[org.osgi.service.event.EventAdmin, GET] so that they may retrieve the Event Admin service and use it. In addition, event sources require TopicPermission[<topic>, PUBLISH] for each topic they want to send events to.

Bundles that need to iterate the handlers registered with the system must be granted ServicePermission[org.osgi.service.event.EventHandler, GET] to retrieve the event handlers from the service registry.

Only a bundle that contains an Event Admin service implementation should be granted ServicePermission[org.osgi.service.event.EventAdmin, REGISTER] to register the event channel admin service.

113.11.3 Security Context During Event Callbacks

During an event notification, the Event Admin service's Protection Domain will be on the stack above the handler's Protection Domain. In the case of a synchronous event, the event publisher's protection domain can also be on the stack.

Therefore, if a handler needs to perform a secure operation using its own privileges, it must invoke the doPrivileged method to isolate its security context from that of its caller.

The event delivery mechanism must not wrap event notifications in a doPrivileged call.

113.12 org.osgi.service.event

Event Admin Package Version 1.4.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.event; version="[1.4,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.event; version="[1.4,1.5)"
```

113.12.1 Summary

- Event - An event.
- EventAdmin - The Event Admin service.
- EventConstants - Defines standard names for EventHandler properties.
- EventHandler - Listener for Events.
- EventProperties - The properties for an Event.
- TopicPermission - A bundle's authority to publish or subscribe to event on a topic.

113.12.2 public class Event

An event. Event objects are delivered to EventHandler services which subscribe to the topic of the event.

Concurrency Immutable

113.12.2.1 public Event(String topic, Map<String, ?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored. If the specified properties is an EventProperties object, then it will be directly used. Otherwise, a copy of the specified properties is made.

□ Constructs an event.

Throws IllegalArgumentException– If topic is not a valid topic name.

Since 1.2

113.12.2.2 public Event(String topic, Dictionary<String, ?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored. A copy of the specified properties is made.

□ Constructs an event.

Throws IllegalArgumentException– If topic is not a valid topic name.

113.12.2.3 public final boolean containsProperty(String name)

name The name of the property.

□ Indicate the presence of an event property. The event topic is present using the property name "event.topics".

Returns true if a property with the specified name is in the event. This property may have a null value. false otherwise.

Since 1.3

113.12.2.4 public boolean equals(Object object)

object The Event object to be compared.

□ Compares this Event object to another object.

An event is considered to be **equal to** another event if the topic is equal and the properties are equal. The properties are compared using the java.util.Map.equals() rules which includes identity comparison for array values.

Returns true if object is a Event and is equal to this object; false otherwise.

113.12.2.5 public final Object getProperty(String name)

name The name of the property to retrieve.

□ Retrieve the value of an event property. The event topic may be retrieved with the property name "event.topics".

Returns The value of the property, or null if not found.

113.12.2.6 public final String[] getPropertyNames()

□ Returns a list of this event's property names. The list will include the event topic property name "event.topics".

Returns A non-empty array with one element per property.

113.12.2.7 public final String getTopic()

□ Returns the topic of this event.

Returns The topic of this event.

113.12.2.8 public int hashCode()

□ Returns a hash code value for this object.

Returns An integer which is a hash code value for this object.

113.12.2.9 public final boolean matches(Filter filter)

filter The filter to test.

□ Tests this event's properties against the given filter using a case sensitive match.

Returns true If this event's properties match the filter, false otherwise.

113.12.2.10 public String toString()

□ Returns the string representation of this event.

Returns The string representation of this event.

113.12.3 public interface EventAdmin

The Event Admin service. Bundles wishing to publish events must obtain the Event Admin service and call one of the event delivery methods.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

113.12.3.1 public void postEvent(Event event)

event The event to send to all listeners which subscribe to the topic of the event.

□ Initiate asynchronous, ordered delivery of an event. This method returns to the caller before delivery of the event is completed. Events are delivered in the order that they are received by this method.

Throws SecurityException– If the caller does not have TopicPermission[topic,PUBLISH] for the topic specified in the event.

113.12.3.2 public void sendEvent(Event event)

event The event to send to all listeners which subscribe to the topic of the event.

□ Initiate synchronous delivery of an event. This method does not return to the caller until delivery of the event is completed.

Throws SecurityException– If the caller does not have TopicPermission[topic,PUBLISH] for the topic specified in the event.

113.12.4 public interface EventConstants

Defines standard names for EventHandler properties.

Provider Type Consumers of this API must not implement this type

113.12.4.1 public static final String BUNDLE = "bundle"

The Bundle object of the bundle relevant to the event. The type of the value for this event property is Bundle.

Since 1.1

113.12.4.2 public static final String BUNDLE_ID = "bundle.id"

The Bundle id of the bundle relevant to the event. The type of the value for this event property is Long.

Since 1.1

113.12.4.3 public static final String BUNDLE_SIGNER = "bundle.signer"

The Distinguished Names of the signers of the bundle relevant to the event. The type of the value for this event property is String or Collection of String.

113.12.4.4 public static final String BUNDLE_SYMBOLICNAME = "bundle.symbolicName"

The Bundle Symbolic Name of the bundle relevant to the event. The type of the value for this event property is String.

113.12.4.5 public static final String BUNDLE_VERSION = "bundle.version"

The version of the bundle relevant to the event. The type of the value for this event property is Version.

Since 1.2

113.12.4.6 public static final String DELIVERY_ASYNC_ORDERED = "async.ordered"

Event Handler delivery quality value specifying the Event Handler requires asynchronously delivered events be delivered in order. Ordered delivery is the default for asynchronously delivered events.

This delivery quality value is mutually exclusive with DELIVERY_ASYNC_UNORDERED. However, if both this value and DELIVERY_ASYNC_UNORDERED are specified for an event handler, this value takes precedence.

See Also EVENT_DELIVERY

Since 1.3

113.12.4.7 public static final String DELIVERY_ASYNC_UNORDERED = "async.unordered"

Event Handler delivery quality value specifying the Event Handler does not require asynchronously delivered events be delivered in order. This may allow an Event Admin implementation to optimize asynchronous event delivery by relaxing ordering requirements.

This delivery quality value is mutually exclusive with DELIVERY_ASYNC_ORDERED. However, if both this value and DELIVERY_ASYNC_ORDERED are specified for an event handler, DELIVERY_ASYNC_ORDERED takes precedence.

See Also EVENT_DELIVERY

Since 1.3

113.12.4.8 public static final String EVENT = "event"

The forwarded event object. Used when rebroadcasting an event that was sent via some other event mechanism. The type of the value for this event property is Object.

113.12.4.9 public static final String EVENT_ADMIN_IMPLEMENTATION = "osgi.event"

The name of the implementation capability for the Event Admin specification

Since 1.4

113.12.4.10 public static final String EVENT_ADMIN_SPECIFICATION_VERSION = "1.4"

The version of the implementation capability for the Event Admin specification

Since 1.4

113.12.4.11 public static final String EVENT_DELIVERY = "event.delivery"

Service Registration property specifying the delivery qualities requested by an Event Handler service.

Event handlers MAY be registered with this property. Each value of this property is a string specifying a delivery quality for the Event handler.

The value of this property must be of type String, String[], or Collection<String>.

See Also DELIVERY_ASYNC_ORDERED, DELIVERY_ASYNC_UNORDERED

Since 1.3

113.12.4.12 public static final String EVENT_FILTER = "event.filter"

Service Registration property specifying a filter to further select Event s of interest to an Event Handler service.

Event handlers MAY be registered with this property. The value of this property is a string containing an LDAP-style filter specification. Any of the event's properties may be used in the filter expression. Each event handler is notified for any event which belongs to the topics in which the handler has expressed an interest. If the event handler is also registered with this service property, then the properties of the event must also match the filter for the event to be delivered to the event handler.

If the filter syntax is invalid, then the Event Handler must be ignored and a warning should be logged.

The value of this property must be of type String.

See Also Event, Filter

113.12.4.13 public static final String EVENT_TOPIC = "event.topics"

Service registration property specifying the Event topics of interest to an Event Handler service.

Event handlers SHOULD be registered with this property. Each value of this property is a string that describe the topics in which the handler is interested. An asterisk ('*') may be used as a trailing wildcard. Event Handlers which do not have a value for this property must not receive events. More precisely, the value of each string must conform to the following grammar:

```
  topic-description := '*' | topic ( '/*' )?
  topic := token ( '/' token )*
```

The value of this property must be of type String, String[], or Collection<String>.

See Also Event

113.12.4.14 public static final String EXCEPTION = "exception"

An exception or error. The type of the value for this event property is Throwable.

113.12.4.15 public static final String EXCEPTION_CLASS = "exception.class"

The name of the exception type. Must be equal to the name of the class of the exception in the event property EXCEPTION. The type of the value for this event property is String.

Since 1.1

113.12.4.16 public static final String EXCEPTION_MESSAGE = "exception.message"

The exception message. Must be equal to the result of calling getMessage() on the exception in the event property EXCEPTION. The type of the value for this event property is String.

113.12.4.17 public static final String EXECPTION_CLASS = "exception.class"

This constant was released with an incorrectly spelled name. It has been replaced by EXCEPTION_CLASS

Deprecated As of 1.1. Replaced by EXCEPTION_CLASS.

113.12.4.18 public static final String MESSAGE = "message"

A human-readable message that is usually not localized. The type of the value for this event property is String.

113.12.4.19 public static final String SERVICE = "service"

A service reference. The type of the value for this event property is ServiceReference.

113.12.4.20 public static final String SERVICE_ID = "service.id"

A service's id. The type of the value for this event property is Long.

113.12.4.21 public static final String SERVICE_OBJECTCLASS = "service.objectClass"

A service's objectClass. The type of the value for this event property is String[].

113.12.4.22 public static final String SERVICE_PID = "service.pid"

A service's persistent identity. The type of the value for this event property is String or Collection of String.

113.12.4.23 public static final String TIMESTAMP = "timestamp"

The time when the event occurred, as reported by System.currentTimeMillis(). The type of the value for this event property is Long.

113.12.5 public interface EventHandler

Listener for Events.

EventHandler objects are registered with the Framework service registry and are notified with an Event object when an event is sent or posted.

EventHandler objects can inspect the received Event object to determine its topic and properties.

EventHandler objects must be registered with a service property EventConstants.EVENT_TOPIC whose value is the list of topics in which the event handler is interested.

For example:

```
String[] topics = new String[] {"com/isv/*"};
Hashtable ht = new Hashtable();
ht.put(EventConstants.EVENT_TOPIC, topics);
context.registerService(EventHandler.class.getName(), this, ht);
```

Event Handler services can also be registered with an EventConstants.EVENT_FILTER service property to further filter the events. If the syntax of this filter is invalid, then the Event Handler must be ignored by the Event Admin service. The Event Admin service should log a warning.

Security Considerations. Bundles wishing to monitor Event objects will require ServicePermission[EventHandler,REGISTER] to register an EventHandler service. The bundle must also have TopicPermission[topic,SUBSCRIBE] for the topic specified in the event in order to receive the event.

See Also Event

Concurrency Thread-safe

113.12.5.1 public void handleEvent(Event event)

event The event that occurred.

□ Called by the EventAdmin service to notify the listener of an event.

113.12.6 public class EventProperties implements Map<String, Object>

The properties for an Event. An event source can create an EventProperties object if it needs to reuse the same event properties for multiple events.

The keys are all of type String. The values are of type Object. The key "event.topics" is ignored as event topics can only be set when an Event is constructed.

Once constructed, an EventProperties object is unmodifiable. However, the values of the map used to construct an EventProperties object are still subject to modification as they are not deeply copied.

Since 1.3

Concurrency Immutable

113.12.6.1 public EventProperties(Map<String, ?> properties)

properties The properties to use for this EventProperties object (may be `null`).

□ Create an EventProperties from the specified properties.

The specified properties will be copied into this EventProperties. Properties whose key is not of type `String` will be ignored. A property with the key "event.topics" will be ignored.

113.12.6.2 public void clear()

□ This method throws UnsupportedOperationException.

Throws `UnsupportedOperationException`– if called.

113.12.6.3 public boolean containsKey(Object name)

name The property name.

□ Indicates if the specified property is present.

Returns `true` If the property is present, `false` otherwise.

113.12.6.4 public boolean containsValue(Object value)

value The property value.

□ Indicates if the specified value is present.

Returns `true` If the value is present, `false` otherwise.

113.12.6.5 public Set<Map.Entry<String, Object>> entrySet()

□ Return the property entries.

Returns A set containing the property name/value pairs.

113.12.6.6 public boolean equals(Object object)

object The `EventProperties` object to be compared.

□ Compares this `EventProperties` object to another object.

The properties are compared using the `java.util.Map.equals()` rules which includes identity comparison for array values.

Returns `true` if `object` is a `EventProperties` and is equal to this object; `false` otherwise.

113.12.6.7 public Object get(Object name)

name The name of the specified property.

□ Return the value of the specified property.

Returns The value of the specified property.

113.12.6.8 public int hashCode()

□ Returns a hash code value for this object.

Returns An integer which is a hash code value for this object.

113.12.6.9 public boolean isEmpty()

□ Indicate if this properties is empty.

Returns `true` If this properties is empty, `false` otherwise.

113.12.6.10 public Set<String> keySet()

□ Return the names of the properties.

Returns The names of the properties.

113.12.6.11 public Object put(String key, Object value)

□ This method throws UnsupportedOperationException.

Throws `UnsupportedOperationException` – if called.

113.12.6.12 public void putAll(Map<? extends String, ? extends Object> map)

□ This method throws UnsupportedOperationException.

Throws `UnsupportedOperationException` – if called.

113.12.6.13 public Object remove(Object key)

□ This method throws UnsupportedOperationException.

Throws `UnsupportedOperationException` – if called.

113.12.6.14 public int size()

□ Return the number of properties.

Returns The number of properties.

113.12.6.15 public String toString()

□ Returns the string representation of this object.

Returns The string representation of this object.

113.12.6.16 public Collection<Object> values()

□ Return the properties values.

Returns The values of the properties.

113.12.7 public final class TopicPermission extends Permission

A bundle's authority to publish or subscribe to event on a topic.

A topic is a slash-separated string that defines a topic.

For example:

```
org/osgi/service/foo/FooEvent/ACTION
```

`TopicPermission` has two actions: `publish` and `subscribe`.

Concurrency Thread-safe

113.12.7.1 public static final String PUBLISH = "publish"

The action string `publish`.

113.12.7.2 public static final String SUBSCRIBE = "subscribe"

The action string `subscribe`.

113.12.7.3 public TopicPermission(String name, String actions)

name Topic name.

actions publish,subscribe (canonical order).

□ Defines the authority to publish and/or subscribe to a topic within the EventAdmin service.

The name is specified as a slash-separated string. Wildcards may be used. For example:

```
org/osgi/service/fooFooEvent/ACTION
com/isv/*
*
```

A bundle that needs to publish events on a topic must have the appropriate TopicPermission for that topic; similarly, a bundle that needs to subscribe to events on a topic must have the appropriate TopicPermssion for that topic.

113.12.7.4 public boolean equals(Object obj)

obj The object to test for equality with this TopicPermission object.

□ Determines the equality of two TopicPermission objects. This method checks that specified TopicPermission has the same topic name and actions as this TopicPermission object.

Returns true if obj is a TopicPermission, and has the same topic name and actions as this TopicPermission object; false otherwise.

113.12.7.5 public String getActions()

□ Returns the canonical string representation of the TopicPermission actions.

Always returns present TopicPermission actions in the following order: publish,subscribe.

Returns Canonical string representation of the TopicPermission actions.

113.12.7.6 public int hashCode()

□ Returns the hash code value for this object.

Returns A hash code value for this object.

113.12.7.7 public boolean implies(Permission p)

p The target permission to interrogate.

□ Determines if the specified permission is implied by this object.

This method checks that the topic name of the target is implied by the topic name of this object. The list of TopicPermission actions must either match or allow for the list of the target object to imply the target TopicPermission action.

```
x/y/*,"publish" -> x/y/z,"publish" is true
*,"subscribe" -> x/y,"subscribe"   is true
*,"publish" -> x/y,"subscribe"     is false
x/y,"publish" -> x/y/z,"publish"   is false
```

Returns true if the specified TopicPermission action is implied by this object; false otherwise.

113.12.7.8 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing TopicPermission objects.

Returns A new PermissionCollection object.

113.13 org.osgi.service.event.annotations

Event Admin Annotations Package Version 1.4.

This package contains annotations that can be used to require the Event Admin implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

113.13.1 Summary

- `RequireEventAdmin` - This annotation can be used to require the Event Admin implementation.

113.13.2 @RequireEventAdmin

This annotation can be used to require the Event Admin implementation. It can be used directly, or as a meta-annotation.

This annotation is applied to several of the Event Admin component property type annotations meaning that it does not normally need to be applied to Declarative Services components which use the Event Admin.

Since 1.4

Retention `CLASS`

Target `TYPE`, `PACKAGE`

113.14 org.osgi.service.event.propertytypes

Event Admin Component Property Types Package Version 1.4.

When used as annotations, component property types are processed by tools to generate Component Descriptions which are used at runtime.

Bundles wishing to use this package at runtime must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.event.propertytypes; version="[1.4,2.0)"
```

113.14.1 Summary

- `EventDelivery` - Component Property Type for the EventConstants.EVENT_DELIVERY service property of an EventHandler service.
- `EventFilter` - Component Property Type for the EventConstants.EVENT_FILTER service property of an EventHandler service.
- `EventTopics` - Component Property Type for the EventConstants.EVENT_TOPIC service property of an EventHandler service.

113.14.2 @EventDelivery

Component Property Type for the EventConstants.EVENT_DELIVERY service property of an EventHandler service.

This annotation can be used on an EventHandler component to declare the value of the EventConstants.EVENT_DELIVERY service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

113.14.2.1 String[] value

□ Service property specifying the Event delivery qualities requested by an EventHandler service.

The supported delivery qualities are:

- EventConstants.DELIVERY_ASYNC_ORDERED
- EventConstants.DELIVERY_ASYNC_UNORDERED}

Returns The requested event delivery qualities.

See Also EventConstants.EVENT_DELIVERY

113.14.3 @EventFilter

Component Property Type for the EventConstants.EVENT_FILTER service property of an EventHandler service.

This annotation can be used on an EventHandler component to declare the value of the EventConstants.EVENT_FILTER service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

113.14.3.1 String value

□ Service property specifying the Event filter to an EventHandler service.

Returns The event filter.

See Also EventConstants.EVENT_FILTER

113.14.4 @EventTopics

Component Property Type for the EventConstants.EVENT_TOPIC service property of an EventHandler service.

This annotation can be used on an EventHandler component to declare the values of the EventConstants.EVENT_TOPIC service property.

See Also Component Property Types

Since 1.4

Retention CLASS

Target TYPE

113.14.4.1 String[] value

□ Service property specifying the Event topics of interest to an EventHandler service.

Returns The event topics.

See Also EventConstants.EVENT_TOPIC

117 Dmt Admin Service Specification

Version 2.0

117.1 Introduction

There are a large number of Device Management standards available today. Starting with the ITU X.700 series in the seventies, SNMP in the eighties and then an explosion of different protocols when the use of the Internet expanded in the nineties. Many device management standards have flourished, and some subsequently withered, over the last decades. Some examples:

- X.700 CMIP
- IETF SNMP
- IETF LDAP
- OMA DM
- Broadband Forum TR-069
- UPnP Forum's Device Management
- IETF NETCONF
- OASIS WS Distributed Management

This heterogeneity of the remote management for OSGi Framework based devices is a problem for device manufacturers. Since there is often no dominant protocol these manufacturers have to develop multiple solutions for different remote management protocols. It is also problematic for device operators since they have to choose a specific protocol but by that choice could exclude a class of devices that do not support that protocol. There is therefore a need to allow the use of multiple protocols at minimal costs.

Almost all management standards are based on hierarchical object models and provide *primitives* like:

- Get and replace values
- Add/Remove instances
- Discovery of value names and instance ids
- Provide notifications

A Device Management standard consists of a *protocol stack* and a number of *object models*. The protocol stack is generic and shared for all object types; the object model describes a specific device's properties and methods. For example, the protocol stack can consist of a set of SOAP message formats and an object model is a `Deployment Unit`. An object model consists of a data model and sometimes a set of functions.

The core problem is that the generic Device Management Tree must be mapped to device specific functions. This specification therefore defines an API for managing a device using general device management concepts but providing an effective plugin model to link the generic tree to the specific device functions.

The API is decomposed in the following packages/functionality:

- `org.osgi.service.dmt` - Main package that provides access to the local Device Management Tree. Access is session based.

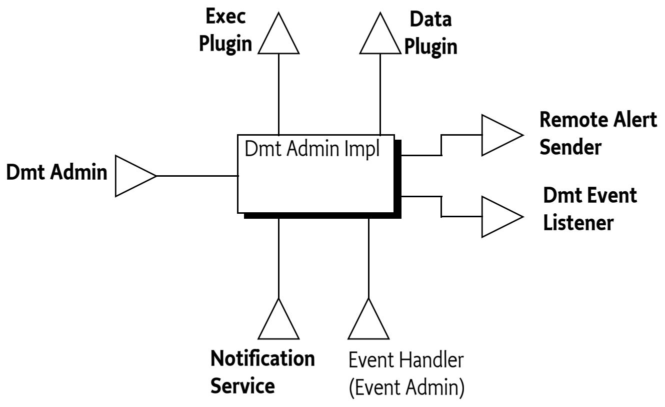
- org.osgi.service.dmt.notification - The notification package provides the capability to send alerts to a management server.
- org.osgi.service.dmt.spi - Provides the capability to register subtree handlers in the Device Management Tree.
- org.osgi.service.dmt.notification.spi - The API to provide the possibility to extend the notification system.
- org.osgi.service.dmt.security - Permission classes.

117.1.1 Entities

- *Device Management Tree* - The Device Management Tree (DMT) is the logical view of manageable aspects of an OSGi Environment, implemented by plugins and structured in a tree with named nodes.
- *Dmt Admin* - A service through which the DMT can be manipulated. It is used by *Local Managers* or by *Protocol Adapters* that initiate DMT operations. The Dmt Admin service forwards selected DMT operations to Data Plugins and execute operations to Exec Plugins; in certain cases the Dmt Admin service handles the operations itself. The Dmt Admin service is a singleton.
- *Dmt Session* - A session groups a set of operations on a sub-tree with optional transactionality and locking. Dmt Session objects are created by the Dmt Admin service and are given to a plugin when they first join the session.
- *Local Manager* - A bundle which uses the Dmt Admin service directly to read or manipulate the DMT. Local Managers usually do not have a principal associated with the session.
- *Protocol Adapter* - A bundle that communicates with a management server external to the device and uses the Dmt Admin service to operate on the DMT. Protocol Adapters usually have a principal associated with their sessions.
- *Meta Node* - Information provided by the node implementer about a node for the purpose of performing validation and providing assistance to users when these values are edited.
- *Multi nodes* - Interior nodes that have a homogeneous set of children. All these children share the same meta node.
- *Plugin* - Services which take the responsibility over a given sub-tree of the DMT: Data Plugin services and Exec Plugin services.
- *Data Plugin* - A Plugin that can create a Readable Data Session, Read Write Data Session, or Transactional Data Session for data operations on a sub-tree for a Dmt Session.
- *Exec Plugin* - A Plugin that can handle execute operations.
- *Readable Data Session* - A plugin session that can only read.
- *Read Write Data Session* - A plugin session that can read and write.
- *Transactional Data Session* - A plugin session that is transactional.
- *Principal* - Represents the optional identity of an initiator of a Dmt Session. When a session has a principal, the Dmt Admin must enforce ACLs and must ignore Dmt Permissions.
- *ACL* - An Access Control List is a set of principals that is associated with permitted operations.
- *Dmt Event* - Information about a modification of the DMT.
- *Dmt Event Listener* - Listeners to Dmt Events. These listeners are services according to the white board pattern.
- *Mount Point* - A point in the DMT where a Plugin or the Dmt Admin service allows other Plugins to have their root.

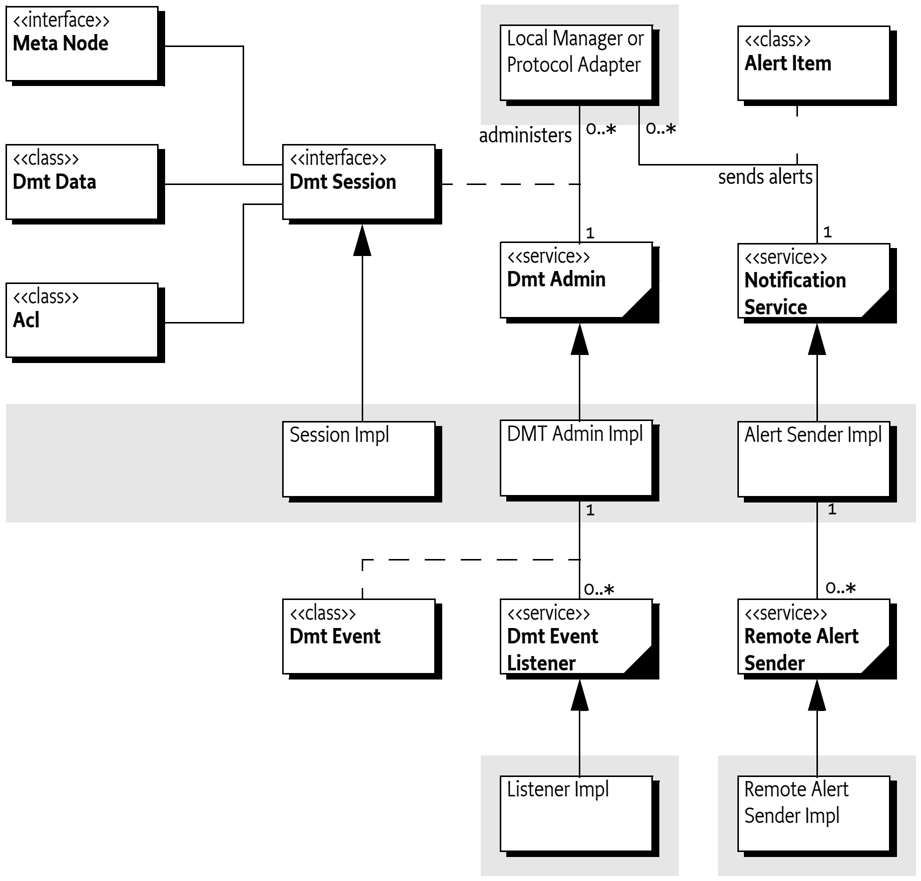
The overall service interaction diagram is depicted in Figure 117.1.

Figure 117.1 Overall Service Diagram



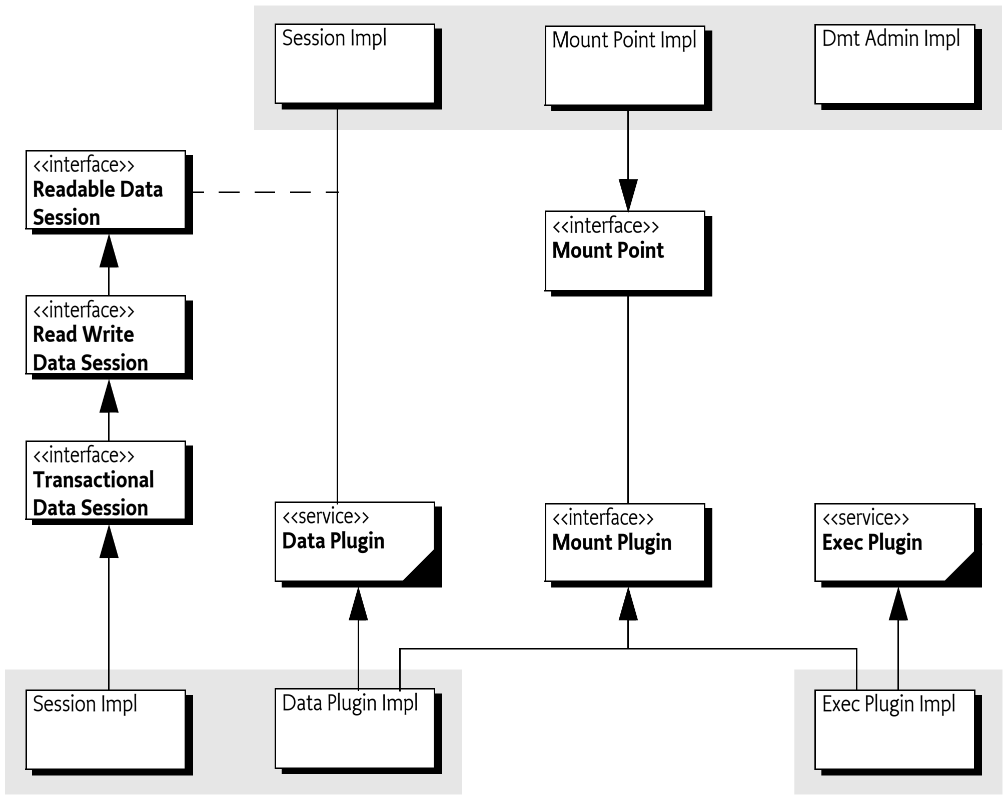
The entities used in the Dmt Admin operations and notifications are depicted in Figure 117.2.

Figure 117.2 Using Dmt Admin service, org.osgi.service.dmt and org.osgi.service.dmt.notification. packages*



Extending the Dmt Admin service with Plugins is depicted in Figure 117.3.

Figure 117.3 Extending the Dmt Admin service, org.osgi.service.dmt.spi package


Session Impl
Mount Point Impl
Dmt Admin Impl
<<interface>> Readable Data Session
<<interface>> Mount Point
<<interface>> Read Write Data Session
<<interface>> Transactional Data Session
<<service>> Data Plugin
<<interface>> Mount Plugin
<<service>> Exec Plugin
Session Impl
Data Plugin Impl
Exec Plugin Impl

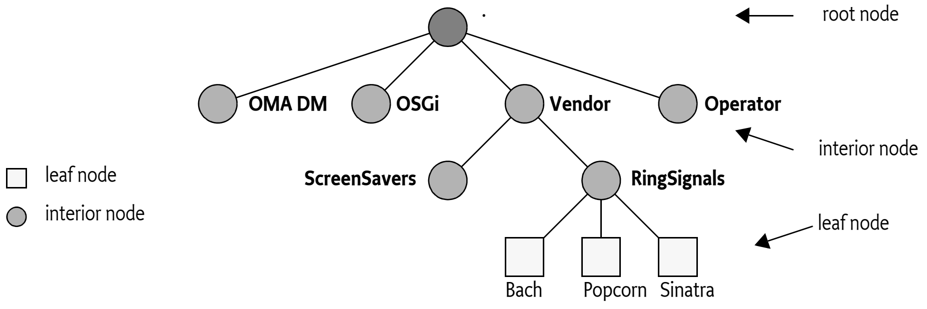
117.2 The Device Management Model

The standard-based features of the DMT model are:

- The Device Management Tree consists of *interior* nodes and *leaf* nodes. Interior nodes can have children and leaf nodes have primitive values.
- All nodes have a set of properties: Name, Title, Format, ACL, Version, Size, Type, Value, and TimeStamp.
- The storage of the nodes is undefined. Nodes typically map to peripheral registers, settings, configuration, databases, etc.
- A node's name must be unique among its siblings.
- Nodes can have Access Control Lists (ACLs), associating operations allowed on those nodes with a particular principal.
- Nodes can have Meta Nodes that describe actual nodes and their siblings.
- Base value types (called *formats* in the standard) are
  - integer
  - long
  - string
  - boolean
  - binary data (multiple types)
  - datetime
  - time

    - float
    - XML fragments
- Leaf nodes in the tree can have default values specified in the meta node.
- Meta Nodes define allowed access operations (Get, Add, Replace, Delete and Exec)

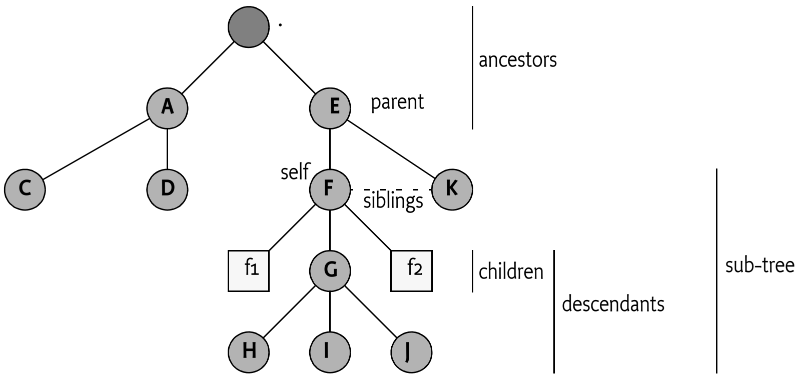
Figure 117.4 *Device Management Tree example*



117.2.1 Tree Terminology

In the following sections, the DMT is discussed frequently. Thus, well-defined terms for all the concepts that the DMT introduces are needed. The different terms are shown in Figure 117.5.

Figure 117.5 *DMT naming, relative to node F*



All terms are defined relative to node F. For this node, the terminology is as follows:

- *URI* - The path consisting of node names that uniquely defines a node, see *The DMT Addressing URI* on page 371.
- *ancestors* - All nodes that are above the given node ordered in proximity. The closest node must be first in the list. In the example, this list is [./E, .]
- *parent* - The first ancestor, in this example this is ./E.
- *children* - A list of nodes that are directly beneath the given node without any preferred ordering. For node F this list is { ./E/F/f1, ./E/F/f2, ./E/F/G }.
- *siblings* - An unordered list of nodes that have the same parent. All siblings must have different names. For F, this is { ./E/K}
- *descendants* - A list of all nodes below the given node. For F this is { ./E/F/f1, ./E/F/G, ./E/F/f2, ./E/F/G/H, ./E/F/G/I, ./E/F/G/J }
- *sub-tree* - The given node plus the list of all descendants. For node F this is { ./E/F, ./E/F/f1, ./E/F/G, ./E/F/f2, ./E/F/G/H, ./E/F/G/I, ./E/F/G/J }

- *overlap* - Two given URIs overlap if they share any node in their sub-trees. In the example, the sub-tree ./E/F and ./E/F/G overlap.
- *data root URI* - A URI which represents the root of a Data Plugin.
- *exec root URI* - A URI which represents the root of an Exec Plugin.
- *Parent Plugin* - A Plugin A is a Parent Plugin of Plugin B if B's root is a in A's sub-tree, this requires a Parent Plugin to at least have one mount point.
- *Child Plugin* - A Plugin A is a Child Plugin of Plugin B if A's root is in B's sub-tree.
- *Scaffold Node* - An ancestor node of a Plugin that is managed by the Dmt Admin service to ensure that all nodes are discoverable by traversing from the root.

117.2.2 Actors

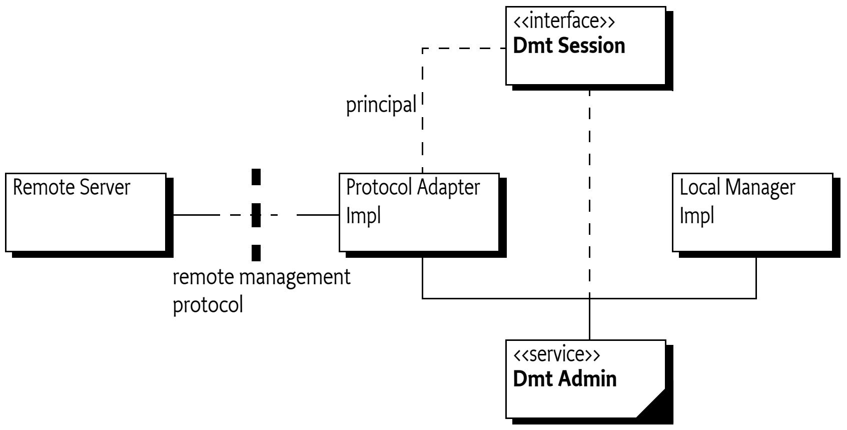
There are two typical users of the Dmt Admin service:

- *Remote manager* - The typical client of the Dmt Admin service is a *Protocol Adapter.* A management server external to the device can issue DMT operations over some management protocol. The protocol to be used is not specified by this specification. For example, OMA DM, TR-069, or others could be used. The protocol operations reach the Framework through the Protocol Adapter, which forwards the calls to the Dmt Admin service in a session. Protocol Adapters should authenticate the remote manager and set the principal in the session. This association will make the Dmt Admin service enforce the ACLs. This requires that the principal is equal to the server name.

  The Dmt Admin service provides a facility to send notifications to the remote manager with the Notification Service.
- *Local Manager* - A bundle which uses the Dmt Admin service to operate on the DMT: for example, a GUI application that allows the end user to change settings through the DMT.

  Although it is possible to manage some aspects of the system through the DMT, it can be easier for such applications to directly use the services that underlie the DMT; many of the management features available through the DMT are also available as services. These services shield the callers from the underlying details of the abstract, and sometimes hard to use DMT structure. As an example, it is more straightforward to use the Monitor Admin service than to operate upon the monitoring sub-tree. The local management application might listen to Dmt Events if it is interested in updates in the tree made by other entities, however, these events do not necessarily reflect the accurate state of the underlying services.

Figure 117.6 *Actors*



117.3 The DMT Admin Service

The Dmt Admin service operates on the Device Management Tree of an OSGi-based device. The Dmt Admin API is loosely modeled after the OMA DM protocol: the operations for `Get`, `Replace`, `Add`, `Delete` and `Exec` are directly available. The Dmt Admin is a singleton service.

Access to the DMT is session-based to allow for locking and transactionality. The sessions are, in principle, concurrent, but implementations that queue sessions can be compliant. The client indicates to the Dmt Admin service what kind of session is needed:

- *Exclusive Update Session* - Two or more updating sessions cannot access the same part of the tree simultaneously. An updating session must acquire an exclusive lock on the sub-tree which blocks the creation of other sessions that want to operate on an overlapping sub-tree.
- *Multiple Readers Session* - Any number of read-only sessions can run concurrently, but ongoing read-only sessions must block the creation of an updating session on an overlapping sub-tree.
- *Atomic Session* - An atomic session is the same as an exclusive update session, except that the session can be rolled back at any moment, undoing all changes made so far in the session. The participants must accept the outcome: rollback or commit. There is no prepare phase. The lack of full two phase commit can lead to error situations which are described later in this document; see *Plugins and Transactions* on page 384.

Although the DMT represents a persistent data store with transactional access and without size limitations, the notion of the DMT should not be confused with a general purpose database. The intended purpose of the DMT is to provide a *dynamic view* of the management state of the device; the DMT model and the Dmt Admin service are designed for this purpose.

117.4 Manipulating the DMT

117.4.1 The DMT Addressing URI

The OMA DM limits URIs to the definition of a URI in [8] *RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax*. The Uri utility classes handles nearly all escaping issues with a number of static methods. All URIs in any of the API methods can use the full Unicode character set. For example, the following URIs as used in Java code are valid URIs for the Dmt Admin service.

```
"./ACME © 2000/A/x"
"./ACME/Address/Street/9C, Avenue St. Drézéry"
```

This strategy has a number of consequences.

- A solidus (`'/' \u002F`) collides with the use of the solidus as separator of the node names. Solidi must therefore be escaped using a reverse solidus (`'\' \u005C`). The reverse solidus must be escaped with a double reverse solidus sequence. The Dmt Admin service must ignore a reverse solidus when it is not followed by a solidus or reverse solidus. The solidus and reverse solidus must not be escaped using the `%00` like escaping defined for URIs. For example, a node that has the name of a MIME type could look like:

  ```
  ./OSGi/mime/application\/png
  ```

  In Java, a reverse solidus must be escaped as well, therefore requiring double reverse solidi:

  ```
  String a = "./OSGi/mime/application\\/png";
  ```

  A literal reverse solidus would therefore require 4 reverse solidi in a Java string.

- The length of a node name is defined to be the length of the byte array that results from UTF-8 encoding a string.

The Uri class provides an encode(String) method to escape a string and a decode(String) method to unescape a string. Though in general the Dmt Admin service implementations should not impose unnecessary constraints on the node name length, it is possible that an implementation runs out of space. In that case it must throw a DmtException URI_TOO_LONG.

Nodes are addressed by presenting a *relative* or *absolute URI* for the requested node. The URI is defined with the following grammar:

```
uri            ::= relative-uri | absolute-uri
absolute-uri   ::= './' relative-uri
relative-uri   ::= segment ( '/' segment )*
segment        ::= (~['/'])*
```

The Uri isAbsoluteUri(String) method makes it simple to find out if a URI is relative or absolute. Relative URIs require a base URI that is for example provided by the session, see *Locking and Sessions* on page 372.

Each node name is appended to the previous ones using a solidus ('/' \u002F) as the separating character. The first node of an absolute URI must be the full stop ('.' \u002E). For example, to access the Bach leaf node in the RingTones interior node from Figure 117.4 on page 369, the URI must be:

```
./Vendor/RingSignals/Bach
```

The URI must be given with the root of the management tree as the starting point. URIs used in the DMT must be treated and interpreted as *case-sensitive*. I.e../Vendor and ./vendor designate two different nodes. The following mandatory restrictions on URI syntax are intended to simplify the parsing of URIs.

The full stop has no special meaning in a node name. That is, sequences like .. do not imply parent node. The isValidUri(String) method verifies that a URI fulfills all its obligations and is valid.

117.4.2 Locking and Sessions

The Dmt Admin service is the main entry point into the DMT, its usage is to create sessions. A simple example is getting a session on a specific sub-tree. Such a session can be created with the getSession(String) method. This method creates an updating session with an exclusive lock on the given sub-tree. The given sub-tree can be a single leaf node, if so desired.

Each session has an ID associated with it which is unique to the machine and is never reused. This id is always greater than 0. The value -1 is reserved as place holder to indicate a situation has no session associated with it, for example an event generated from an underlying service. The URI argument addresses the sub-tree root. If null, it addresses the root of the DMT. All nodes can be reached from the root, so specifying a session root node is not strictly necessary but it permits certain optimizations in the implementations.

If the default exclusive locking mode of a session is not adequate, it is possible to specify the locking mode with the getSession(String,int) and getSession(String,String,int) method. These methods supports the following locking modes:

- LOCK_TYPE_SHARED - Creates a *shared session*. It is limited to read-only access to the given sub-tree, which means that multiple sessions are allowed to read the given sub-tree at the same time.
- LOCK_TYPE_EXCLUSIVE - Creates an *exclusive session*. The lock guarantees full read-write access to the tree. Such sessions, however, cannot share their sub-tree with any other session. This type of lock requires that the underlying implementation supports Read Write Data Sessions.
- LOCK_TYPE_ATOMIC - Creates an *atomic session* with an exclusive lock on the sub-tree, but with added transactionality. Operations on such a session must either succeed together or fail togeth-

er. This type of lock requires that the underlying implementation supports Transactional Data Sessions. If the Dmt Admin service does not support transactions, then it must throw a Dmt Exception with the FEATURE_NOT_SUPPORTED code. If the session accesses data plugins that are not transactional in write mode, then the Dmt Admin service must throw a Dmt Exception with the TRANSACTION_ERROR code. That is, data plugins can participate in a atomic sessions as long as they only perform read operations.

The Dmt Admin service must lock the sub-tree in the requested mode before any operations are performed. If the requested sub-tree is not accessible, the getSession(String,int), getSession(String,String,int), or getSession(String) method must block until the sub-tree becomes available. The implementation can decide after an implementation-dependent period to throw a Dmt Exception with the SESSION_CREATION_TIMEOUT code.

As a simplification, the Dmt Admin service is allowed to lock the entire tree irrespective of the given sub-tree. For performance reasons, implementations should provide more fine-grained locking when possible.

Persisting the changes of a session works differently for exclusive and atomic sessions. Changes to the sub-tree in an atomic session are not persisted until the commit() or close() method of the session is called. Changes since the last transaction point can be rolled back with the rollback() method.

The commit() and rollback() methods can be called multiple times in a session; they do not close the session. The open, commit(), and rollback() methods all establish a *transaction point*. The rollback operation cannot roll back further than the last transaction point.

Once a fatal error is encountered (as defined by the DmtException isFatal() method), all successful changes must be rolled back automatically to the last transaction point. Non-fatal errors do not rollback the session. Any error/exception in the commit or rollback methods invalidates and closes the session. This can happen if, for example, the mapping state of a plugin changes that has its plugin root inside the session's sub-tree.

Changes in an exclusive session are persisted immediately after each separate operation. Errors do not roll back any changes made in such a session.

Due to locking and transactional behavior, a session of any type must be closed once it is no longer used. Locks must always be released, even if the close() method throws an exception.

Once a session is closed no further operations are allowed and manipulation methods must throw a Dmt Illegal State Exception when called. Certain information methods like for example getState() and getRootUri() can still be called for logging or diagnostic purposes. This is documented with the Dmt Session methods.

The close() or commit() method can be expected to fail even if all or some of the individual operations were successful. This failure can occur due to multi-node constraints defined by a specific implementation. The details of how an implementation specifies such constraints is outside the scope of this specification.

Events in an atomic session must only be sent at commit time.

117.4.3 Associating a Principal

Protocol Adapters must use the getSession(String,String,int) method which features the principal as the first parameter. The principal identifies the external entity on whose behalf the session is created. This server identification string is determined during the authentication process in a way specific to the management protocol.

For example, the identity of the OMA DM server can be established during the handshake between the OMA DM agent and the server. In the simpler case of OMA CP protocol, which is a one-way protocol based on WAP Push, the identity of the principal can be a fixed value.

117.4.4 Relative Addressing

All DMT operation methods are found on the session object. Most of these methods accept a *relative* or *absolute* URI as their first parameter: for example, the method isLeafNode(String). This URI is absolute or relative to the sub-tree with which the session is associated. For example, if the session is opened on:

```
./Vendor
```

then the following URIs address the Bach ring tone:

```
RingTones/Bach
./Vendor/RingTones/Bach
```

Opening the session with a null URI is identical to opening the session at the root. But the absolute URI can be used to address the Bach ring tone as well as a relative URI.

```
./Vendor/RingTones/Bach
Vendor/RingTones/Bach
```

If the URI specified does not correspond to a legitimate node in the tree, a Dmt Exception must be thrown. The only exception to this rule is the isNodeUri(String) method that can verify if a node is actually valid. The getMetaNode(String) method must accept URIs to non-existing nodes if an applicable meta node is available; otherwise it must also throw a Dmt Exception.

117.4.5 Creating Nodes

The methods that create interior nodes are:

- createInteriorNode(String) - Create a new interior node using the default meta data. If the principal does not have Replace access rights on the parent of the new node then the session must automatically set the ACL of the new node so that the creating server has Add, Delete and Replace rights on the new node.
- createInteriorNode(String,String) - Create a new interior node. The meta data for this new node is identified by the second argument, which is a URI *identifying* an OMA DM Device Description Framework (DDF) file, this does not have to be a valid location. It uses a format like org.osgi/1.0/LogManagementObject. This meta node must be consistent with any meta information from the parent node.
- createLeafNode(String) - Create a new leaf node with a default value.
- createLeafNode(String,DmtData) - Create a leaf node and assign a value to the leaf-node.
- createLeafNode(String,DmtData,String) - Create a leaf node and assign a value for the node. The last argument is the MIME type, which can be null.

For a node to be created, the following conditions must be fulfilled:

- The URI of the new node has to be a valid URI.
- The principal of the Dmt Session, if present, must have ACL Add permission to add the node to the parent. Otherwise, the caller must have the necessary permission.
- All constraints of the meta node must be verified, including value constraints, name constraints, type constraints, and MIME type constraints. If any of the constraints fail, a Dmt Exception must be thrown with an appropriate code.

117.4.6 Node Properties

A DMT node has a number of runtime properties that can be set through the session object. These properties are:

- *Title* - (String) A human readable title for the object. The title is distinct from the node name. The title can be set with setNodeTitle(String,String) and read with getNodeTitle(String). This specification does not define how this information is localized. This property is optional depending on the implementation that handles the node.

- *Type* -(String) The MIME type, as defined in [9] *MIME Media Types*, of the node's value when it is a leaf node. The type of an interior node is a string identifying a DDF type. These types can be set with setNodeType(String,String) and read with getNodeType(String).
- *Version* - (int) Version number, which must start at 0, incremented after every modification (for both a leaf and an interior node) modulo 0x10000. Changes to the value or any of the properties (including ACLs), or adding/deleting nodes, are considered changes. The getNodeVersion(String) method returns this version; the value is read-only. In certain cases, the underlying data structure does not support change notifications or makes it difficult to support versions. This property is optional depending on the node's implementation.
- *Size* - (int) The size measured in bytes is read-only and can be read with getNodeSize(String). Not all nodes can accurately provide this information.
- *Time Stamp* -(Date) Time of the last change in version. The getNodeTimestamp(String) returns the time stamp. The value is read only. This property is optional depending on the node's implementation.
- *ACL* - The Access Control List for this and descendant nodes. The property can be set with setNodeAcl(String,Acl) and obtained with getNodeAcl(String).

If a plugin that does not implement an optional property is accessed, a Dmt Exception with the code FEATURE_NOT_SUPPORTED must be thrown.

117.4.7 Setting and Getting Data

Values are represented as DmtData objects, which are immutable. The are acquired with the getNodeValue(String) method and set with the setNodeValue(String,DmtData) method.

DmtData objects are dynamically typed by an integer enumeration. In OMA DM, this integer is called the *format* of the data value. The format of the DmtData class is similar to the type of a variable in a programming language, but the word *format* is used here. The available data formats are listed in the following table.

Table 117.1 *Data Formats*

Format Type	Java Type	Format Name	Constructor	Get	Description
FORMAT_BASE64	byte[]	base64	DmtData(byte[],boolean)	getBase64()	Binary type that must be encoded with base 64, see [10] *RFC 3548 The Base16, Base32, and Base64 Data Encodings.*
FORMAT_BINARY	byte[]	binary	DmtData(byte[]) DmtData(byte[],boolean)	getBinary()	A byte array. The DmtData object is created with the constructor. The byte array can only be acquired with the method.
FORMAT_BOOLEAN	boolean	boolean	DmtData(boolean)	getBoolean()	Boolean. There are two constants for this type: • FALSE_VALUE • TRUE_VALUE
FORMAT_DATE	String	date	DmtData(String,int)	getString() getDate()	A Date (no time). Syntax defined in [13] *XML Schema Part 2: Datatypes Second Edition* as the date type.

Format Type	Java Type	Format Name	Constructor	Get	Description
FORMAT_DATE_TIME	String	dateTime	DmtData(Date)	getDateTime()	A Date object representing a point in time.
FORMAT_FLOAT	float	float	DmtData(float)	getFloat()	Float
FORMAT_INTEGER	int	integer	DmtData(int)	getInt()	Integer
FORMAT_LONG	long	long	DmtData(long)	getLong()	Long
FORMAT_NODE	Object	NODE	DmtData(Object)	getNode()	A DmtData object can have a format of FORMAT_NODE. This value is returned from a MetaNode getFormat() method if the node is an interior node or for a data value when the Plugin supports complex values.
FORMAT_NULL					No valid data is available. DmtData objects with this format cannot be constructed; the only instance is the DmtData NULL_VALUE constant.
FORMAT_RAW_BINARY	byte[]	<custom>	DmtData(String,byte[])	getRawBinary()	A raw binary format is always created with a format name. This format name allows the creator to define a proprietary format. The format name is available from the getFormatName() method, which has predefined values for the standard formats.
FORMAT_RAW_STRING	String	<custom>	DmtData(String,String)	getRawString()	A raw string format is always created with a format name. This format name allows the creator to define a proprietary format. The format name is available from the getFormatName() method, which has predefined values for the standard formats.
FORMAT_STRING	String	string	DmtData(String)	getString()	String
FORMAT_TIME	String	time	DmtData(String,int)	getString()	Time of Day. Syntax defined in [13] *XML Schema Part 2: Datatypes Second Edition* as the time type.

Format Type	Java Type	Format Name	Constructor	Get	Description
FORMAT_XML	String	xml	DmtData(String,int)	getXml()	A string containing an XML fragment. It can be obtained with. The validity of the XML must not be verified by the Dmt Admin service.

117.4.8 Complex Values

The OMA DM model prescribes that only leaf nodes have primitive values. This model maps very well to remote managers. However, when a manager is written in Java and uses the Dmt Admin API to access the tree, there are often unnecessary conversions from a complex object, to leaf nodes, and back to a complex object. For example, an interior node could hold the current GPS position as an OSGi Position object, which consists of a longitude, latitude, altitude, speed, and direction. All these objects are Measurement objects which consist of value, error, and unit. Reading such a Position object through its leaf nodes only to make a new Position object is wasting resources. It is therefore that the Dmt Admin service also supports *complex values* as a supplementary facility.

If a complex value is used then the leaves must also be accessible and represent the same semantics as the complex value. A manager unaware of complex values must work correctly by only using the leaf nodes. Setting or getting the complex value of an interior node must be identical to setting or getting the leaf nodes.

Accessing a complex value requires Get access to the node and all its descendants. Setting a complex value requires Replace access to the interior node. Replacing a complex value must only generate a single Replace event.

Trying to set or get a complex value on an interior node that does not support complex values must throw a Dmt Exception with the code FEATURE_NOT_SUPPORTED.

117.4.9 Nodes and Types

The node's type can be set with the setNodeType(String,String) method and acquired with getNodeType(String). The namespaces for the types differ for interior and leaf nodes. A leaf node is typed with a MIME type and an interior node is typed with a DDF Document URI. However, in both cases the Dmt Admin service must not verify the syntax of the type name.

The createLeafNode(String,DmtData,String) method takes a MIME type as last argument that will type the leaf node. The MIME type reflects how the data of the node should be *interpreted*. For example, it is possible to store a GIF and a JPEG image in a DmtData object with a FORMAT_BINARY format. Both the GIF and the JPEG object share the same *format*, but will have MIME types of image/jpg and image/gif respectively. The Meta Node provides a list of possible MIME types.

The createInteriorNode(String,String) method takes a DDF Document URI as the last argument that will type the interior node. This specification defines the DDF Document URIs listed in the following table for interior nodes that have a particular meaning in this specification.

Table 117.2 Standard Interior Node Types

Interior Node Type	Description
DDF_SCAFFOLD	Scaffold nodes are automatically generated nodes by the Dmt Admin service to provide the children node names so that Plugins are reachable from the root. See *Scaffold Nodes* on page 385.
DDF_MAP	MAP nodes define a key -> value mapping construct using the node name (key) and the node value (value). See *MAP Nodes* on page 413.

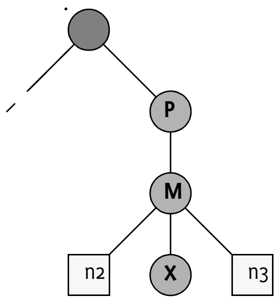
Interior Node Type	Description
DDF_LIST	LIST nodes use the node name to maintain an index in a list. See *LIST Nodes* on page 411.

117.4.10 Deleting Nodes

The deleteNode(String) method on the session represents the Delete operation. It deletes the sub-tree of that node. This method is applicable to both leaf and interior nodes. Nodes can be deleted by the Dmt Admin service in any order. The root node of the session cannot be deleted.

For example, given Figure 117.7, deleting node P must delete the nodes ./P, ./P/ M, ./P/M/X, ./P/M/n2 and ./P/M/n3 in any order.

Figure 117.7 DMT node and deletion



117.4.11 Copying Nodes

The copy(String,String,boolean) method on the DmtSession object represents the Copy operation. A node is completely copied to a new URI. It can be specified with a boolean if the whole sub-tree (true) or just the indicated node is copied.

The ACLs must not be copied; the new access rights must be the same as if the caller had created the new nodes individually. This restriction means that the copied nodes inherit the access rights from the parent of the destination node, unless the calling principal does not have Replace rights for the parent. See *Creating Nodes* on page 374 for details.

117.4.12 Renaming Nodes

The renameNode(String,String) method on the DmtSession object represents the Rename operation, which replaces the node name. It requires permission for the Replace operation. The root node for the current session can not be renamed.

117.4.13 Execute

The execute(String,String) and execute(String,String,String) methods can *execute* a node. Executing a node is intended to be used when a problem is hard to model as a set of leaf nodes. This can be related to synchronization issues or data manipulation. The execute methods can provide a correlator for a notification and an opaque string that is forwarded to the implementer of the node.

Execute operations can not take place in a read only session because simultaneous execution could make conflicting changes to the tree.

117.4.14 Closing

When all the changes have been made, the session must be closed by calling the close() method on the session. The Dmt Admin service must then finalize, clean up, and release any locks. For atomic sessions, the Dmt Admin service must automatically commit any changes that were made since the last transaction point.

A session times out and is invalidated after an extended period of inactivity. The exact length of this period is not specified, but is recommended to be at least 1 minute and at most 24 hours. All methods of an invalidated session must throw an Dmt Illegal State Exception after the session is invalidated.

A session's state is one of the following: STATE_CLOSED, STATE_INVALID or STATE_OPEN, as can be queried by the getState() call. The invalid state is reached either after a fatal error case is encountered or after the session is timed out. When an atomic session is invalidated, it is automatically rolled back to the last transaction point of the session.

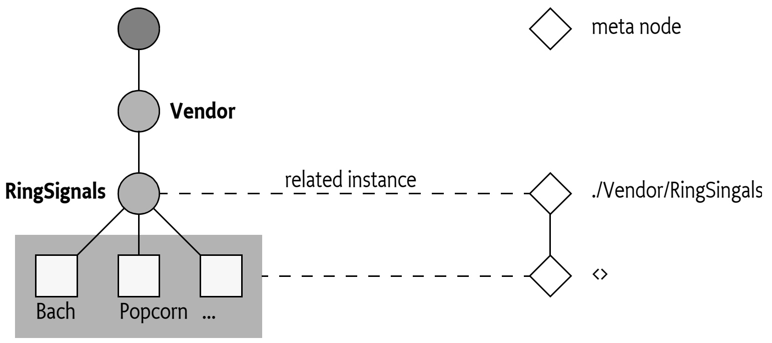
117.5 Meta Data

The getMetaNode(String) method returns a MetaNode object for a given URI. This node is called the *meta node*. A meta node provides information about nodes.

Any node can optionally have a meta node associated with it. The one or more nodes that are described by the meta nodes are called the meta node's *related instances*. A meta node can describe a singleton-related instance, or it can describe all the children of a given parent if it is a *multi-node*. That is to say, meta nodes can exist without an actual instance being present. In order to retrieve the meta node of a multi-node any name can be used.

For example, if a new ring tone, Grieg, was created in Figure 117.8 it would be possible to get the Meta Node for ./Vendor/RingSignals/Grieg before the node was created. This is usually the case for multi nodes. The model is depicted in Figure 117.8.

Figure 117.8 *Nodes and meta nodes*



A URI is generally associated with the same Meta Node. The getMetaNode(String) should return the same meta node for the same URI except in the case of *Scaffold Nodes* on page 385. As the ownership of scaffold nodes can change from the Dmt Admin service to the Parent Plugin service, or from a Parent Plugin to a Child Plugin, the Meta Node can change as well.

The last segment of the URI to get a Meta Node can be any valid node name, for example, instead of Grieg it would have been possible to retrieve the same Meta Node with the name ./Vendor/RingSignals/0, ./Vendor/RingSignals/anyName, ./Vendor/RingSignals/<>, etc.

The actual meta data can come from two sources:

- *Dmt Admin* - Each Dmt Admin service likely has a private meta data repository. This meta data is placed in the device in a proprietary way.
- *Plugins* - Plugins can carry meta nodes and provide these to the Dmt Admin service by implementing the getMetaNode(String[]) method. If a plugin returns a non-null value, the Dmt Admin service must use that value, possibly complemented by its own metadata for elements not provided by the plugin.

The MetaNode interface supports methods to retrieve read-only meta data. The following sections describes this meta-data in more detail.

117.5.1 Operations

The can(int) method provide information as to whether the associated node can perform the given operation. This information is only about the capability; it can still be restricted in runtime by ACLs and permissions.

For example, if the can(MetaNode.CMD_EXECUTE) method returns true, the target object supports the Execute operation. That is, calling the execute(String,String) method with the target URI is possible.

The can(int) method can take the following constants as parameters:

- CMD_ADD
- CMD_DELETE
- CMD_EXECUTE
- CMD_GET
- CMD_REPLACE

For example:

```
void foo( DmtSession session, String nodeUri) {
    MetaNode    meta = session.getMetaNode(nodeUri);
  if ( meta !=null && meta.can(MetaNode.CMD_EXECUTE) )
        session.execute(nodeUri,"foo" );
}
```

117.5.2 Scope

The scope is part of the meta information of a node. It provides information about what the life cycle role is of the node. The getScope() method on the Meta Node provides this information. The value of the scope can be one of the following:

- DYNAMIC - Dynamic nodes are intended to be created and deleted by a management system or an other controlling source. This does not imply that it actually is possible to add new nodes and delete nodes, the actions can still allow or deny this. However, in principle nodes that can be added or deleted have the DYNAMIC scope. The LIST and MAP nodes, see *OSGi Object Modeling* on page 407, always have DYNAMIC scope.
- PERMANENT - Permanent nodes represent an entity in the system. This can be a network interface, a device description, etc. Permanent nodes in general map to an object in an object oriented language. Despite their name, PERMANENT nodes can appear and disappear, for example the plugging in of a USB device might create a new PERMANENT node. Generally, the Plugin roots map to PERMANENT nodes.
- AUTOMATIC - Automatic nodes map in general to nodes that are closely tied to the parent. They are similar to fields of an object in an object oriented language. They cannot be deleted or added.

For example, a node representing the Battery can never be deleted because it is an intrinsic part of the device; it will therefore be PERMANENT. The Level and number of ChargeCycle nodes will be AUTOMATIC. A new ring tone is dynamically created by a manager and is therefore DYNAMIC.

117.5.3 Description and Default

- getDescription() - (String) A description of the node. Descriptions can be used in dialogs with end users: for example, a GUI application that allows the user to set the value of a node. Localization of these values is not defined.

- getDefault() - (DmtData) A default data value.

117.5.4 Validation

The validation information allows the runtime system to verify constraints on the values; it also allows user interfaces to provide guidance.

A node does not have to exist in the DMT in order to have meta data associated with it. Nodes may exist that have only partial meta data, or no metadata, associated with them. For each type of metadata, the default value to assume when it is omitted is described in MetaNode.

117.5.5 Data Types

A leaf node can be constrained to a certain format and one of a set of MIME types.

- getFormat() - (int) The required type. This type is a logical OR of the supported formats.
- getRawFormatNames() - Return an array of possible raw format names. This is only applicable when the getFormat() returns the FORMAT_RAW_BINARY or FORMAT_RAW_STRING formats. The method must return null otherwise.
- getMimeTypes() - (String[]) A list of MIME types for leaf nodes or DDF types for interior nodes. The Dmt Admin service must verify that the actual type of the node is part of this set.

117.5.6 Cardinality

A meta node can constrain the number of *siblings* (i.e., not the number of children) of an interior or leaf node. This constraint can be used to verify that a node must not be deleted, because there should be at least one node left on that level (isZeroOccurrenceAllowed()), or to verify that a node cannot be created, because there are already too many siblings (getMaxOccurrence()).

If the cardinality of a meta node is more than one, all siblings must share the same meta node to prevent an invalid situation. For example, if a node has two children that are described by different meta nodes, and any of the meta nodes has a cardinality >1, that situation is invalid.

For example, the ./Vendor/RingSignals/<> meta node (where <> stands for any name) could specify that there should be between 0 and 12 ring signals.

- getMaxOccurrence() - (int) A value greater than 0 that specifies the maximum number of instances for this node.
- isZeroOccurrenceAllowed() - (boolean) Returns true if zero instances are allowed. If not, the last instance must not be deleted.

117.5.7 Matching

The following methods provide validation capabilities for leaf nodes.

- isValidValue(DmtData) - (DmtData) Verify that the given value is valid for this meta node.
- getValidValues() - (DmtData[]) A set of possible values for a node, or null otherwise. This can for example be used to give a user a set of options to choose from.

117.5.8 Numeric Ranges

Numeric leaf nodes (format must be FORMAT_INTEGER, FORMAT_LONG, or FORMAT_FLOAT) can be checked for a minimum and maximum value.

Minimum and maximum values are inclusive. That is, the range is [getMin(),getMax()]. For example, if the maximum value is 5 and the minimum value is -5, then the range is [-5,5]. This means that valid values are -5,-4,-3,-2... 4, 5.

- getMax() - (double) The value of the node must be less than or equal to this maximum value.

- getMin() - (double) The value of the node must be greater than or equal to this minimum value.

If no meta data is provided for the minimum and maximum values, the meta node must return the Double.MIN_VALUE, and Double.MAX_VALUE respectively.

117.5.9 Name Validation

The meta node provides the following name validation facilities for both leaf and interior nodes:

- isValidName(String) - (String) Verifies that the given name matches the rules for this meta node.
- getValidNames() - (String[]) An array of possible names. A valid name for this node must appear in this list.

117.5.10 User Extensions

The Meta Node provides an extension mechanism; each meta node can be associated with a number of properties. These properties are then interpreted in a proprietary way. The following methods are used for user extensions:

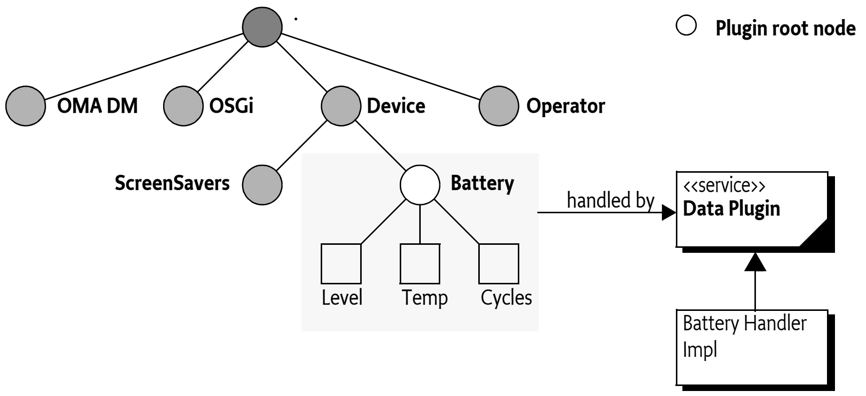
- getExtensionPropertyKeys() - Returns an array of key names that can be provided by this meta node.
- getExtensionProperty(String) - Returns the value of an extension property.

For example, a manufacturer could use a regular expression to validate the node names with the isValidName(String) method. In a web based user interface it is interesting to provide validity checking in the browser, however, in such a case the regular expression string is required. This string could then be provided as a user extension under the key x-acme-regex-javascript.

117.6 Plugins

The Plugins take the responsibility of handling DMT operations within certain sub-trees of the DMT. It is the responsibility of the Dmt Admin service to forward the operation requests to the appropriate plugin. The only exceptions are the ACL manipulation commands. ACLs must be enforced by the Dmt Admin service and never by the plugin. The model is depicted in Figure 117.9.

Figure 117.9 Device Management Tree example



Plugins are OSGi services. The Dmt Admin service must dynamically map and unmap the plugins, acting as node handler, as they are registered and unregistered. Service properties are used to specify the sub-tree that the plugin can manage as well as mount points that it provides to *Child Plugins*; plugins that manage part of the Plugin's sub-tree.

For example, a plugin related to Configuration Admin handles the sub-tree which stores configuration data. This sub-tree could start at ./OSGi/Configuration. When the client wants to add a new configuration object to the DMT, it must issue an Add operation to the ./OSGi/Configuration node. The Dmt Admin service then forwards this operation to the configuration plugin. The plugin maps the request to one or more method calls on the Configuration Admin service. Such a plugin can be a simple proxy to the Configuration Admin service, so it can provide a DMT view of the configuration data store.

There are two types of Dmt plugins: *data plugins* and *exec plugins*. A data plugin is responsible for handling the sub-tree retrieval, addition and deletion operations, and handling of meta data, while an exec plugin handles a node execution operation.

117.6.1 Data Sessions

Data Plugins must participate in the Dmt Admin service sessions. A Data Plugin provider must therefore register a Data Plugin service. Such a service can create a session for the Dmt Admin service when the given sub-tree is accessed by a Dmt Session. If the associated Dmt Session is later closed, the Data Session will also be closed. Three types of sessions provide different capabilities. Data Plugins do not have to implement all session types; if they choose not to implement a session type they can return null.

- *Readable Data Session* - Must always be supported. It provides the basic read-only access to the nodes and the close() method. The Dmt Admin service uses this session type when the lock mode is LOCK_TYPE_SHARED for the Dmt Session. Such a session is created with the plugin's openReadOnlySession(String[],DmtSession), method which returns a ReadableDataSession object.
- *Read Write Data Session* - Extends the Readable Data Session with capabilities to modify the DMT. This is used for Dmt Sessions that are opened with LOCK_TYPE_EXCLUSIVE. Such a session is created with the plugin's openReadWriteSession(String[],DmtSession) method, which returns a ReadWriteDataSession object.
- *Transactional Data Session* - Extends the Read Write Data Session with commit and rollback methods so that this session can be used with transactions. It is used when the Dmt Session is opened with lock mode LOCK_TYPE_ATOMIC. Such a session is created with the plugin's openAtomicSession(String[],DmtSession) method, which returns a TransactionalDataSession object.

117.6.2 URIs and Plugins

The plugin Data Sessions do not use a simple string to identify a node as the Dmt Session does. Instead the URI parameter is a String[]. The members of this String[] are the different segments. The first node after the root is the second segment and the node name is the last segment. The different segments require escaping of the solidus ('/' \u002F) and reverse solidus ('\' \u005C).

The reason to use String[] objects instead of the original string is to reduce the number times that the URI is parsed. The entry String objects, however, are still escaped. For example, the URI ./A/B/image\/jpg gives the following String[]:

```
{ ".", "A", "B", "image\/jpg" }
```

A plugin can assume that the path is validated and can be used directly.

117.6.3 Associating a sub-tree

Each plugin is associated with one or more DMT sub-trees. The top node of a sub-tree is called the *plugin root*. The plugin root is defined by a service registration property. This property is different for exec plugins and data plugins:

- DATA_ROOT_URIS - (String+) A sequence of *data URI*, defining a plugin root for data plugins.

- EXEC_ROOT_URIS - (String+) A sequence of *exec URI*, defining a plugin root for exec plugins.

If the Plugin modifies these service properties then the Dmt Admin service must reflect these changes as soon as possible. The reason for the different properties is to allow a single service to register both as a Data Plugin service as well as an Exec Plugin service.

Data and Exec Plugins live in independent trees and can fully overlap. However, an Exec Plugin can only execute a node when the there exists a valid node at the corresponding node in the Data tree. That is, to be able to execute a node it is necessary that isNodeUri(String) would return true.

For example, a data plugin can register itself in its activator to handle the sub-tree ./Dev/Battery:

```
public void start(BundleContext context) {
  Hashtable ht = new Hashtable();
  ht.put(Constants.SERVICE_PID, "com.acme.data.plugin");
  ht.put( DataPlugin.DATA_ROOT_URIS, "./Dev/Battery");
  context.registerService(
        DataPlugin.class.getName(),
        new BatteryHandler(context);
        ht );
}
```

If this activator was executed, an access to ./Dev/Battery must be forwarded by the Dmt Admin service to this plugin via one of the data session.

117.6.4 Synchronization with Dmt Admin Service

The Dmt Admin service can, in certain cases, detect that a node was changed without the plugin knowing about this change. For example, if the ACL is changed, the version and timestamp must be updated; these properties are maintained by the plugin. In these cases, the Dmt Admin service must open a ReadableDataSession and call nodeChanged(String[]) method with the changed URI.

117.6.5 Plugin Meta Data

Plugins can provide meta data; meta data from the Plugin must take precedence over the meta data of the Dmt Admin service. If a plugin provides meta information, the Dmt Admin service must verify that an operation is compatible with the meta data of the given node.

For example if the plugin reports in its meta data that the ./A leaf node can only have the text/plain MIME type, the createLeafNode(String) calls must not be forwarded to the Plugin if the third argument specifies any other MIME type. If this contract between the Dmt Admin service and the plugin is violated, the plugin should throw a Dmt Exception METADATA_MISMATCH.

117.6.6 Plugins and Transactions

For the Dmt Admin service to be transactional, transactions must be supported by the data plugins. This support is not mandatory in this specification, and therefore the Dmt Admin service has no transactional guarantees for atomicity, consistency, isolation or durability. The DmtAdmin interface and the DataPlugin (or more specifically the data session) interfaces, however, are designed to support Data Plugin services that are transactional. Exec plugins need not be transaction-aware because the execute method does not provide transactional semantics, although it can be executed in an atomic transaction.

Data Plugins do not have to support atomic sessions. When the Dmt Admin service creates a Transactional Data Session by calling openAtomicSession(String[],DmtSession) the Data Plugin is allowed to return null. In that case, the plugin does not support atomic sessions. The caller receives a Dmt Exception.

Plugins must persist any changes immediately for Read Write Data Sessions. Transactional Data Sessions must delay changes until the commit() method is called, which can happen multiple times during a session. The opening of an atomic session and the commit() and rollback() methods all establish a *transaction point*. Rollback can never go further back than the last transaction point.

- commit() - Commit any changes that were made to the DMT but not yet persisted. This method should not throw an Exception because other Plugins already could have persisted their data and can no longer roll it back. The commit method can be called multiple times in an open session, and if so, the commit must make persistent the changes since the last transaction point.
- rollback() - Undo any changes made to the sub-tree since the last transaction point.
- close() - Clean up and release any locks. The Dmt Admin service must call the commit methods before the close method is called. A Plugin must not perform any persistency operations in the close method.

The commit(), rollback(), and close() plugin data session methods must all be called in reverse order of that in which Plugins joined the session.

If a Plugin throws a fatal exception during an operation, the Dmt Session must be rolled back immediately, automatically rolling back all data plugins, as well as the plugins that threw the fatal Dmt Exception. The fatality of an Exception can be checked with the Dmt Exception isFatal() method.

If a plugin throws a non-fatal exception in any method accessing the DMT, the current operation fails, but the session remains open for further commands. All errors due to invalid parameters (e.g. non-existing nodes, unrecognized values), all temporary errors, etc. should fall into this category.

A rollback of the transaction can take place due to any irregularity during the session. For example:

- A necessary Plugin is unregistered or unmapped
- A fatal exception is thrown while calling a plugin
- Critical data is not available
- An attempt is made to breach the security

Any Exception thrown during the course of a commit() or rollback() method call is considered fatal, because the session can be in a half-committed state and is not safe for further use. The operation in progress should be continued with the remaining Plugins to achieve a *best-effort* solution in this limited transactional model. Once all plugins have been committed or rolled back, the Dmt Admin service must throw an exception, specifying the cause exception(s) thrown by the plugin(s), and should log an error.

117.6.7 Side Effects

Changing a node's value will have a side effect of changing the system. A plugin can also, however, cause state changes with a get operation. Sometimes the pattern to use a get operation to perform a state changing action can be quite convenient. The get operation, however, is defined to have no side effects. This definition is reflected in the session model, which allows the DMT to be shared among readers. Therefore, plugins should refrain from causing side effects for read-only operations.

117.6.8 Copying

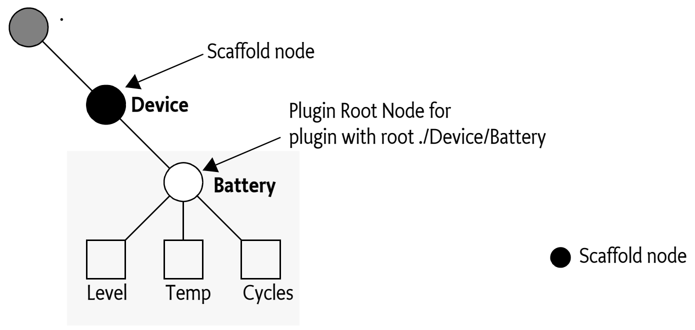
Plugins do not have to support the copy operation. They can throw a Dmt Exception with a code FEATURE_NOT_SUPPORTED. In this case, the Dmt Admin service must do the copying node by node. For the clients of the Dmt Admin service, it therefore appears that the copy method is always supported.

117.6.9 Scaffold Nodes

As Plugins can be mapped anywhere into the DMT it is possible that a part of the URI has no corresponding Plugin, such a plugin would not be *reachable* unless the intermediate nodes were provided. A program that would try to discover the DMT would not be able to find the registered Plugins as the intermediate nodes would not be discoverable.

These intermediate nodes that will make all plugins reachable must therefore be provided by the Dmt Admin service, they are called the *scaffold nodes*. The only purpose of the scaffold nodes is to allow every node to be discovered when the DMT is traversed from the root down. Scaffold nodes are provided both for Data Plugins as well as Exec Plugins as well as for Child Plugins that are mounted inside a Parent Plugin, see *Sharing the DMT* on page 388. In Figure 117.10 the Device node is a scaffold node because there is no plugin associated with it. The Dmt Admin service must, however, provide the Battery node as child node of the Device node.

Figure 117.10 *Scaffold Nodes*



A scaffold node is always an interior node and has limited functionality, it must have a type of DDF_SCAFFOLD. It has no value, it is impossible to add or delete nodes to it, and the methods that are allowed for a scaffold node are specified in the following table.

Table 117.3 *Supported Scaffold Node Methods*

Method	Description
getNodeAcl(String)	Must inherit from the root node.
getChildNodeNames(String)	Answer the child node names such that plugin's in the sub-tree are reachable.
getMetaNode(String)	Provides the Meta Node defined in Table 117.4
getNodeSize(String)	Must throw a DmtException COMMAND_NOT_ALLOWED
getNodeTitle(String)	null
getNodeTimestamp(String)	Time first created
getNodeType(String)	DDF_SCAFFOLD
isNodeUri(String)	true
isLeafNode(String)	false
getNodeVersion(String)	Away returns 0
copy(String,String,boolean)	Not allowed for a single scaffold node as nodeUri, if the recurse parameter is false the DmtException COMMAND_NOT_ALLOWED

Any other operations must throw a DmtException with error code COMMAND_NOT_ALLOWED. The scope of a scaffold node is always PERMANENT. Scaffold nodes must have a Meta Node provided by the Dmt Admin service. This Meta Node must act as defined in the following table.

Table 117.4 *Scaffold Meta Node Supported Methods*

Method	Description
can(int)	CMD_GET
getDefault()	null

Method	Description
getDescription()	null
getFormat()	FORMAT_NODE
getMax()	Double.MAX_VALUE
getMaxOccurrence()	1
getMimeTypes()	DDF_SCAFFOLD
getMin()	Double.MIN_VALUE
getRawFormatNames()	null
getScope()	PERMANENT
getValidNames()	null
getValidValues()	null
isLeaf()	false
isValidName(String)	true
isValidValue(DmtData)	false
isZeroOccurrenceAllowed()	true

If a Plugin is registered then it is possible that a scaffold node becomes a Data Plugin root node. In that case the node and the Meta Node must subsequently be provided by the Data Plugin and can thus become different. Scaffold nodes are virtual, there are therefore no events associated with the life cycle of a scaffold node.

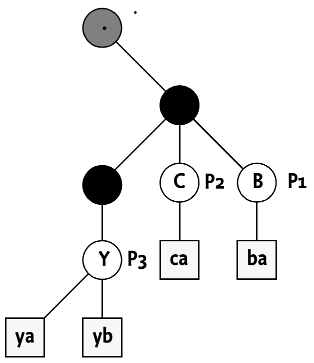
For example, there are three plugins registered:

```
URI         Plugin  Children
./A/B       P1      ba
./A/C       P2      ca
./A/X/Y     P3      ya,yb
```

In this example, node B, C, and Y are the plugin roots of the different plugins. As there is no plugin the manage node A and X these must be provided by the Dmt Admin service. In this example, the child names returned from each node are summarized as follows:

```
Node        Children        Provided by
.           { A }           Dmt Admin (scaffold node)
A           { X, C, B }     Dmt Admin (scaffold node)
B           { ba }          P1
C           { ca }          P2
X           { Y  }          Dmt Admin (scaffold node)
Y           { ya, yb }      P3
```

Figure 117.11 *Example Scaffold Nodes*



117.7 Sharing the DMT

The Dmt Admin service provides a model to integrate the management of the myriad of components that make up an OSGi device. This integration is achieved by sharing a single namespace: the DMT. Sharing a single namespace requires rules to prevent conflicts and to resolve any conflicts when Plugins register with plugin roots that overlap. It also requires rules for the Dmt Admin service when nodes are accessed for which there is no Plugin available.

This section defines the management of overlapping plugins through the *mount points*, places where a Parent Plugin can allow a Child Plugin to take over.

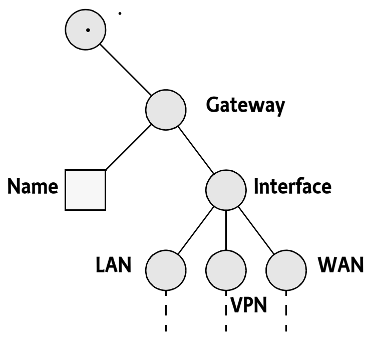
117.7.1 Mount Points

With multiple plugins the DMT is a *shared namespace*. Sharing requires rules to ensure that conflicts are avoided and when they occur, can be resolved in a consistent way. The most powerful and flexible model is to allow general overlapping. However, in practice this flexibility comes at the cost of ordering issues and therefore timing dependent results. A best practice is therefore to strictly control the points where the DMT can be extended for both Data and Exec Plugins.

A *mount point* is such a place. A Dmt Admin service at start up provides virtual mount points anywhere in the DMT and provides scaffold nodes for any intermediate nodes between the root of the DMT and the Plugin's root URI. Once a Plugin is mounted it is free to use its sub-tree (the plugin root and any ancestors) as it sees fit. However, this implies that the Plugin must implement the full subtree. In reality, many object models use a pattern where the different levels in the object model map to different domains.

For example, an Internet Gateway could have an object model where the general information, like the name, vendor, etc. is stored in the first level but any attached interfaces are stored in the sub-tree. However, It is highly unlikely that the code that handles the first level with the general information is actually capable of handling the details of, for example, the different network interfaces. It is actually likely that these network interfaces are dynamic. A Virtual Private Network (VPN) can add virtual network interfaces on demand. Such a could have the object model depicted in Figure 117.12.

Figure 117.12 Data Modeling



Forcing these different levels to be implemented by the same plugin violates one of the primary rules of modularity: *cohesion*. Plugins forced to handle all aspects become complex and hard to maintain. A Plugin like the one managing the Gateway node could provide its own Plugin mechanism but that would force a lot of replication and is error prone. For this reason, the Dmt Admin service allows a Plugin to provide *mount points* inside its sub-tree. A Plugin can specify that it has mount points by registering a MOUNT_POINTS service property (the constant is defined both in DataPlugin and ExecPlugin but have the same constant value). The type of this property must be String+, each string specifies a mount point. Each mount point is specified as a URI that is relative from the plugin root. That is, when the plugin root is ./A/B and the mount point is specified as C then the absolute URI of the mount point is ./A/B/C.

A Plugin that has mount points acts as a *Parent Plugin* to a number of *Child Plugins*. In the previous example, the LAN, VPN, and WAN nodes, can then be provided by separate Child Plugins even though the Gateway/Name node is provided by the Parent Plugin. In this case, the mount points are children of the Interface node.

A mount point can be used by a number of child plugins. In the previous example, there was a Child Plugin for the LAN node, the VPN node, and the WAN node. This model has the implicit problem that it requires coordination to ensure that their names are unique. Such a coordination between independent parties is complicated and error prone. Its is therefore possible to force the Dmt Admin service to provide unique names for these nodes, see *Shared Mount Points* on page 390.

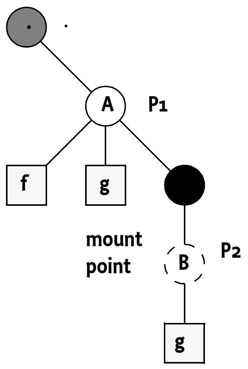
A Parent Plugin is not responsible for any scaffolding nodes to make its Child Plugins reachable. However, Dmt Admin may assume that a Plugin Root node always exists and may not provide a scaffold node on the Plugin Root. A Plugin is recommended to always provide the Plugin Root node to make its Child Plugins reachable. When a Parent Plugin provides the nodes to its mount points, the nodes should be the correct interior nodes to make its Child Plugins reachable.

For example, the following setup of plugins:

```
Plugin      Plugin Root     Mount Points
P1          ./A             X/B
P2          ./A/X/B
```

This setup is depicted in Figure 117.13.

Figure 117.13 *Example Scaffold Nodes For Child Plugin*



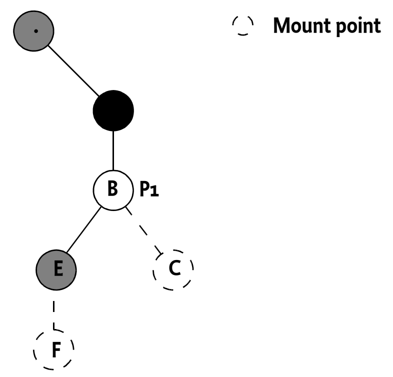
If the child node names are requested for the ./A node then the plugin P1 is asked for the child node names and must return the names [f,g]. However, if plugin P2 is mapped then the Dmt Admin service must add the scaffold node name that makes this plugin reachable from that level, the returned set must therefore be [f, g, X].

117.7.2 Parent Plugin

If a Plugin is registered with mount points then it is a *Parent Plugin*. A Parent Plugin must register with a single plugin root URI, that is the DATA_ROOT_URIS or EXEC_ROOT_URIS service properties must contain only one element. A Parent Plugin is allowed to be a Data and Exec Plugin at the same time. If a Parent Plugin is registered with multiple plugin root URIs then the Dmt Admin service must log an error and ignore the registration of such a Parent Plugin. A Parent Plugin can in itself also be a Child Plugin.

For example, a Plugin P1 that has a plugin root of ./A/B and provides a mount point at ./A/B/C and ./A/B/E/F. as depicted in Figure 117.14.

Figure 117.14 Example Mount Points



Registering such a Plugin would have to register the following service properties to allow the example configuration of the DMT:

```
dataRootUris    ./A/B
mountPoints     [ C, E/F ]
```

117.7.3 Shared Mount Points

Mount points can be shared between different Plugins. In the earlier example about the Gateway the Interface node contained a sub-tree of network interfaces. It is very likely in such an example that the Plugins for the VPN interface will be provided by a different organization than the WAN and LAN network interfaces. However, all these network interface plugins must share a single parent node, the Interface node, under which they would have to mount. Sharing therefore requires a prior agreement and a naming scheme.

The naming scheme is defined by using the number sign ('#' \u0023) to specify a *shared mount point*. A plugin root that ends with the number sign, for example ./A/B/#, indicates that it is willing to get any node under node B, leaving the naming of that node up to the Dmt Admin service. Shared mount points cannot overlap with normal mount points, the first one will become mapped and subsequent ones are in error, they are incompatible with each other. A Parent Plugin must specify a mount point explicitly as a shared mount point by using the number sign at the end of the mount point's relative URI.

A plugin is compatible with other plugins if all other plugins specify a shared mount point to the same URI. It is compatible with its Parent Plugin if the child's plugin root and the mount point are either shared or not.

The Dmt Admin service must provide a name for a plugin root that identifies a shared mount point such that every Plugin on that mount point has a unique integer name for that node level. The integer name must be >= 1. The name must be convertible to an int with the static Integer parseInt(String) method.

A management system in general requires permanent links to nodes. It is therefore necessary to choose the same integer every time a plugin is mapped to a shared mount point. A Child Plugin on a shared mount point must therefore get a permanent integer node name when it registers with a Persistent ID (PID). That is, it registers with the service property service.pid. The permanent link is then coupled to the PID and the bundle id since different bundles must be able to use the same PID. If a Plugin is registered with multiple PIDs then the first one must be used. Since permanent links can stay around for a long time implementations must strive to not reuse these integer names.

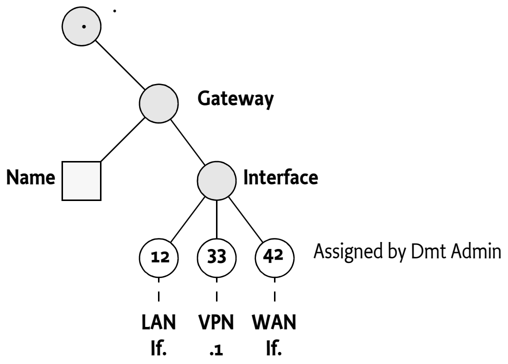
If no PID is provided then the Dmt Admin service must choose a new number that has not been used yet nor matches any persistently stored names that are currently not registered.

The Gateway example would require the following Plugin registrations:

```
Root URI                 Mount Points    Plugin     Role
./Gateway                [Interface/#]   Gateway    Parent
./Gateway/Interface/#    []              WAN If.    Child
./Gateway/Interface/#    []              LAN If.    Child
./Gateway/Interface/#    []              VPN.1      Child
```

This setup is depicted in Figure 117.15.

Figure 117.15 *Mount Point Sharing*



The Meta Node for a Node on the level of the Mount Point can specify either an existing Plugin or it can refer to a non-existing node. If the node exists, the corresponding Plugin must provide the Meta Node. If the node does not exist, the Dmt Admin service must provide the Meta Node. Such a Meta Node must provide the responses as specified in Table 117.4.

Table 117.5 *Shared Mount Point Meta Node Supported Methods*

Method	Description
can(int)	CMD_GET
getDefault()	null
getDescription()	null
getFormat()	FORMAT_NODE
getMax()	Double.MAX_VALUE
getMaxOccurrence()	Integer.MAX_VALUE
getMimeTypes()	null
getMin()	Double.MIN_VALUE
getRawFormatNames()	null
getScope()	The scope will depend on the Parent
getValidNames()	null
getValidValues()	null
isLeaf()	false
isValidName(String)	name >=1 && name < Integer.MAX_VALUE
isValidValue(DmtData)	false
isZeroOccurrenceAllowed()	true

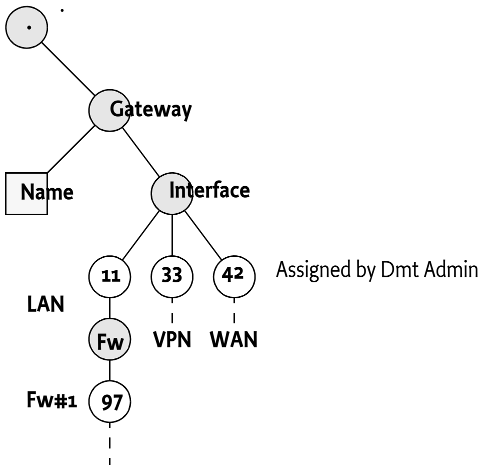
A URI can cross multiple mount points, shared and unshared. For example, if a network interface could be associated with a number of firewall rules then it is possible to register a URI on the designated network interface that refers to the Firewall rules. For the previous example, a Plugin could register a Firewall if the following registrations were done:

```
Root URI                      Mount Points    Plugin   Parent   Name
```

```
./Gateway                     [Interface/#]   Gw
./Gateway/Interface/#         [Fw/#]          WAN If.  Gw      11
./Gateway/Interface/#         []              LAN If.  Gw      33
./Gateway/Interface/#         []              VPN.1    Gw      42
./Gateway/Interface/11/Fw/#  []              Fw.1     WAN If.  97
```

This example DMT is depicted in Figure 117.16.

Figure 117.16 Mount Point Multiple Sharing



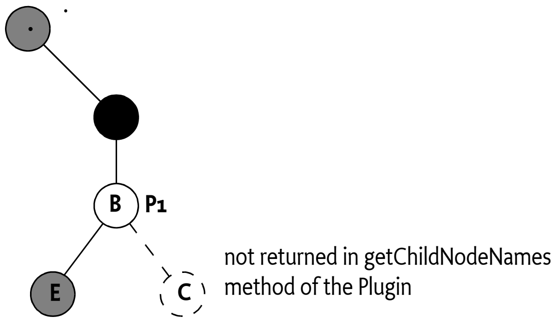
117.7.4 Mount Points are Excluded

Mount nodes are logically not included in the sub-tree of a Plugin. The Dmt Admin service must never ask any information from/about a Mount Point node to its Parent Plugin. A Parent Plugin must also not return the name of a mount point in the list of child node names, the Mount Point and its subtree is logically excluded from the sub-tree. For the Dmt Admin service an unoccupied mount point is a node that does not exist. Its name, must only be discoverable if a Plugin has actually mounted the node. The Dmt Admin service must ensure that the names of the mounted Plugins are included for that level.

In the case of shared mount points the Dmt Admin service must provide the children names of all registered Child Plugins that share that node level.

For example, a Plugin P1 registered with the plugin root of ./A/B, having two leaf nodes E, and a mount point C must not return the name C when the child node names for node B are requested. This is depicted in Figure 117.17. The Dmt Admin service must ensure that C is returned in the list of names when a Plugin is mounted on that node.

Figure 117.17 Example Exclusion



117.7.5 Mapping a Plugin

A Plugin is not stand alone, its validity can depend on other Plugins. Invalid states make it possible that a Plugin is either *mapped* or *unmapped*. When a Plugin is mapped it is available in the DMT and when it is unmapped it is not available. Any registration, unregistration, or modification of its services properties of a Plugin can potentially alter the mapped state of any related Plugin. A plugin becomes *eligible* for mapping when it is registered.

A plugin can have multiple roots. However, the mapping is described as if there is a single plugin root. Plugins with multiple roots must be treated as multiple plugins that can each independently be mapped or unmapped depending on the context.

If no Parent Plugin is available, the Dmt Admin service must act as a virtual Parent Plugin that allows mount points anywhere in the tree where there is no mapped plugin yet.

When a Plugin becomes eligible then the following assertions must be valid for that Plugin to become mapped:

- If it has one or more mount points then
  - It must have at most one Data and/or Exec Root URI.
  - None of its mount points must overlap.
  - Any already mapped Child Plugins must be compatible with its mount points.
- If no mount points are specified then there must be no Child Plugins already registered.
- The plugin root must be compatible with the corresponding parent's mount point. When a Parent Plugin is available, the plugin root must match exactly to the absolute URI of the parent's mount point.
- The plugin root must be compatible with any other plugins on that mount point.

If either of these assertions fail then the Dmt Admin service must log an error and ignore the registered Plugin, it must not become mapped. If, through the unregistration or modification of the service properties, the assertions can become valid then the Dmt Admin service must retry mapping the Plugin so that it can become available in the DMT. Any mappings and unmappings that affect nodes that are in the sub-tree of an active session must abort that session with a CONCURRENT_ACCESS exception.

When there are errors in the configuration then the ordering will define which plugins are mapped or not. Since this is an error situation no ordering is defined between conflicting plugins.

For example, a number of Plugins are registered in the given order:

```
Plugin Root     Children    Mount Points    Plugin
./A/B           E           C               P1
./A/B/C                                     P2
./A/B/D                                     P3
```

The first Plugin P1 will be registered immediately without problems. It has only a single plugin root as required by the fact that it is a Parent Plugin (it has a mount point). There are no Child Plugins yet so it is impossible to have a violation of the mount points. As there is no Parent Plugin registered, the Dmt Admin service will map plugin P1 and automatically provide the scaffold node A.

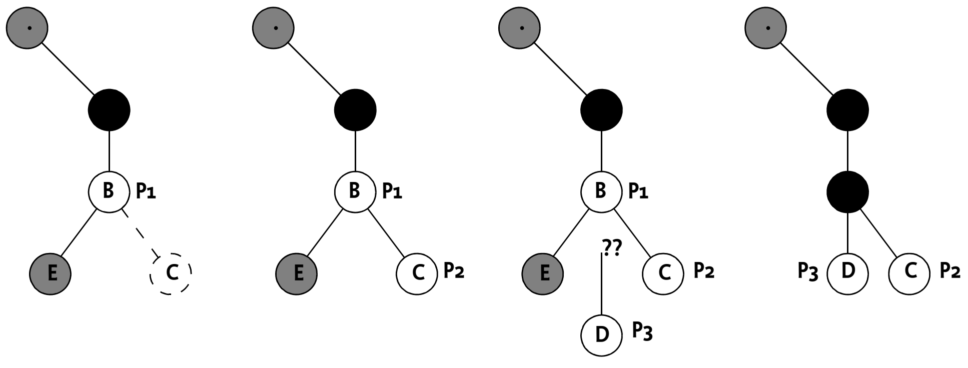
When Plugin P2 is registered its plugin root maps to a mount point in Plugin P1. As P2 is not a Parent Plugin it is only necessary that it has no Child Plugins. As it has no Child Plugins, the mapping will succeed.

Plugin P3 cannot be mapped because the Parent Plugin is P1 but P1 does not provide a mount point for P3's plugin root ./A/B/D.

If, at a later time P1 is unregistered then the Dmt Admin service must map plugin P3 and leave plugin P2 mapped. This sequence of action is depicted in Figure 117.18.

If plugin P1 becomes registered again at a later time it can then in its turn not be mapped as there would be a child plugin (P3) that would not map to its mount point.

Figure 117.18 *Plugin Activation*



P1 Registered and mapped

P2 registered and mapped

P3 is registered but cannot be mapped

P1 is unregistered mapping P3

117.7.6 Mount Plugins

In *Mapping a Plugin* on page 393 it is specified that a Plugin can be *mapped* or not. The mapped state of a Plugin can change depending on other plugins that are registered and unregistered. Plugins require in certain cases to know:

- What is the name of their root node if they mount on a shared mount point.
- What is the mapping state of the Plugin.

To find out these details a Plugin can implement the MountPlugin interface; this is a mixin interface, it is not necessary to register it as MountPlugin service. The Dmt Admin service must do an instanceof operation on Data Plugin services and Exec Plugin services to detect if they are interested in the mount point information.

The Mount Point interface is used by the Dmt Admin service to notify the Plugin when it becomes mapped and when it becomes unmapped. The Plugin will be informed about each plugin root separately.

The Mount Plugin specifies the following methods that are called synchronously:

- mountPointAdded(MountPoint) - The Dmt Admin service must call this method after it has mapped a plugin root. From this point on the given mount point provides the actual path until the mountPointRemoved(MountPoint) is called with an equal object. The given Mount Point can be used to post events.
- mountPointRemoved(MountPoint) - The Dmt Admin service must call this method after it has unmapped the given mount point. This method must always be called when a plugin root is unmapped, even if this is caused by the unregistration of the plugin.

As the mapping and unmapping of a plugin root can happen any moment in time a Plugin that implements the Mount Plugin interface must be prepared to handle these events at any time on any thread.

The MountPoint interface has two separate responsibilities:

- *Path* - The path that this Mount Point is associated with. This path is a plugin root of the plugin. This path is identical to the Plugin's root except when it is mounted on a shared mount point; in that case the URI ends in the name chosen by the Dmt Admin service. The getMountPath() method provides the path.

- *Events* - Post events about the given sub-tree that signal internal changes that occur outside a Dmt Session. The Dmt Admin service must treat these events as they were originated from modifications to the DMT. That is, they need to be forwarded to the Event Admin as well as the Dmt Listeners. For this purpose there are the postEvent(String,String[],Dictionary) and postEvent(String,String[],String[],Dictionary) methods.

For example, a Data Plugin monitoring one of the batteries registers with the following service properties:

```
dataRootURIs                 "./Device/Battery/#"
```

The Device node is from a Parent Plugin that provided the shared mount point. The Battery Plugin implements the MountPlugin interface so it gets called back when it is mapped. This will cause the Dmt Admin service to call the mountPointAdded(MountPoint) method on the plugin. In this case, it will get just one mount point, the mount point for its plugin root. If the Dmt Admin service would have assigned the Battery Plugin number 101 then the getMountPath() would return:

```
[ ".", "Device", "Battery", "101" ]
```

As the Plugin monitors the charge state of the battery it can detect a significant change. In that case it must send an event to notify any observers. The following code shows how this could be done:

```
@Component( properties="dataRootURIs=./Device/Battery/#",
                provide=DataPlugin.class)
public class Battery implements DataPlugin, MountPlugin {
    Timer          timer;
    volatile float charge;
    TimerTask      task;

    public void mountPointsAdded(final MountPoint[] mountPoints){
        task = new TimerTask() {
            public void run() {
                float next = measure();
                if (Math.abs(charge - next) > 0.2) {
                    charge = next;
                    mountPoints[0].postEvent(DmtConstants.EVENT_TOPIC_REPLACED,
                        new String[] { "Charge" }, null);
                }
            }
        };
        timer.schedule(task, 1000);
    }

    public void mountPointsRemoved(MountPoint[] mountPoints){
            task.cancel();
            task = null;
    }
    ... // Other methods
}
```

117.8 Access Control Lists

Each node in the DMT can be protected with an *access control list*, or *ACL*. An ACL is a list of associations between *Principal* and *Operation*:

- *Principal* - The identity that is authorized to use the associated operations. Special principal is the wildcard ('*' \u002A); the operations granted to this principal are called the *global permissions*. The global permissions are available to all principals.
- *Operation* - A list of operations: ADD, DELETE, GET, REPLACE, EXECUTE.

DMT ACLs are defined as strings with an internal syntax in [1] *OMA DM-TND v1.2 draft*. Instances of the ACL class can be created by supplying a valid OMA DM ACL string as its parameter. The syntax of the ACL is presented here in shortened form for convenience:

```
acl          ::= ( acl-entry ( '&' acl-entry)* )
acl-entry    ::= command '=' ( principals | '*' )
principals   ::= principal ( '+' principal )*
principal    ::= -['=' '&' '*' '+' '\t' '\n' '\r']+
```

The principal name should only use printable characters according to the OMA DM specification.

```
command      ::= 'Add' | 'Delete' | 'Exec'| 'Get' | 'Replace'
```

White space between tokens is not allowed.

Examples:

```
Add=*&Replace=*&Get=*

Add=www.sonera.fi-8765&Delete=www.sonera.fi-8765& «
Replace=www.sonera.fi-8765+321_ibm.com&Get=*
```

The Acl(String) constructor can be used to construct an ACL from an ACL string. The toString() method returns a String object that is formatted in the specified form, also called the canonical form. In this form, the principals must be sorted alphabetically and the order of the commands is:

```
 ADD,   DELETE,   EXEC,   GET,   REPLACE
```

The Acl class is immutable, meaning that a Acl object can be treated like a string, and that the object cannot be changed after it has been created.

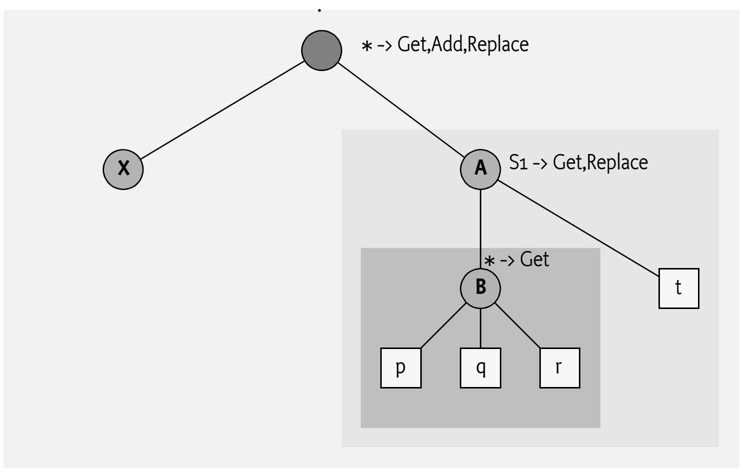
ACLs must only be verified by the Dmt Admin service when the session has an associated principal.

ACLs are properties of nodes. If an ACL is *not set* (i.e. contains no commands nor principals), the *effective* ACL of that node must be the ACL of its first ancestor that has a non-empty ACL. This effective ACL can be acquired with the getEffectiveNodeAcl(String) method. The root node of DMT must always have an ACL associated with it. If this ACL is not explicitly set, it should be set to Add=*&Get=*&Replace=*.

This effect is shown in Figure 117.19. This diagram shows the ACLs set on a node and their effect (which is shown by the shaded rectangles). Any principal can get the value of p, q and r, but they cannot replace, add or delete the node. Node t can only be read and replaced by principal S1.

Node X is fully accessible to any authenticated principal because the root node specifies that all principals have Get, Add and Replace access (*->G,A,R).

Figure 117.19 ACL inheritance



The definition and example demonstrate the access rights to the properties of a node, which includes the value.

Changing the ACL property itself has different rules. If a principal has Replace access to an interior node, the principal is permitted to change its own ACL property *and* the ACL properties of all its child nodes. Replace access on a leaf node does not allow changing the ACL property itself.

In the previous example, only principal S1 is authorized to change the ACL of node B because it has Replace permission on node B's parent node A.

Figure 117.20 ACLs for the ACL property

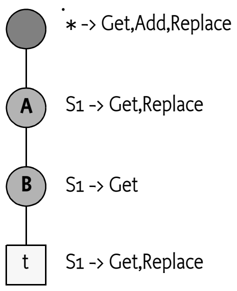


Figure 117.20 demonstrates the effect of this rule with an example. Server S1 can change the ACL properties of all interior nodes. A more detailed analysis:

- *Root* - The root allows all authenticated principals to access it. The root is an interior node so the Replace permission permits the change of the ACL property.
- *Node A* - Server S1 has Replace permission and node A is an interior node so principal S1 can modify the ACL.
- *Node B* - Server S1 has no Replace permission for node B, but the parent node A of node B grants principal S1 Replace permission, and S1 is therefore permitted to change the ACL.
- *Node t* - Server S1 must not be allowed to change the ACL of node t, despite the fact that it has Replace permission on node t. For leaf nodes, permission to change an ACL is defined by the Replace permission in the parent node's ACL. This parent, node B, has no such permission set and thus, access is denied.

The following methods provide access to the ACL property of the node.

- getNodeAcl(String) - Return the ACL for the given node, this method must not take any ACL inheritance into account. The ACL may be null if no ACL is set.
- getEffectiveNodeAcl(String) - Return the effective ACL for the given node, taking any inheritance into account.
- setNodeAcl(String,Acl) - Set the node's ACL. The ACL can be null, in which case the effective permission must be derived from an ancestor. The Dmt Admin service must call nodeChanged(String[]) on the data session with the given plugin to let the plugin update any timestamps and versions.

The Acl class maintains the permissions for a given principal in a bit mask. The following permission masks are defined as constants in the Acl class:

- ADD
- DELETE
- EXEC
- GET
- REPLACE

The class features methods for getting permissions for given principals. A number of methods allow an existing ACL to be modified while creating a new ACL.

- addPermission(String,int) - Return a new Acl object where the given permissions have been added to permissions of the given principal.
- deletePermission(String,int) - Return a new Acl object where the given permissions have been removed from the permissions of the given principal.
- setPermission(String,int) - Return a new Acl object where the permissions of the given principal are overwritten with the given permissions.

Information from a given ACL can be retrieved with:

- getPermissions(String) - (int) Return the combined permission mask for this principal.
- getPrincipals() - (String[]) Return a list of principals (String objects) that have been granted permissions for this node.

Additionally, the isPermitted(String,int) method verifies if the given ACL authorizes the given permission mask. The method returns true if all commands in the mask are allowed by the ACL.

For example:

```
Acl acl = new Acl("Get=S1&Replace=S1");

if ( acl.isPermitted("S1", Acl.GET+Acl.REPLACE ))
    ... // will execute

if ( acl.isPermitted(
    "S1", Acl.GET+Acl.REPLACE+Acl.ADD ))
    ... // will NOT execute
```

117.8.1 Global Permissions

Global permissions are indicated with the '*' and the given permissions apply to all principals. Processing the global permissions, however, has a number of non-obvious side effects:

- Global permissions can be retrieved and manipulated using the special '*' principal: all methods of the Acl class that have a principal parameter also accept this principal.

- Global permissions are automatically granted to all specific principals. That is, the result of the getPermissions or isPermitted methods will be based on the OR of the global permissions and the principal-specific permissions.
- If a global permission is revoked, it is revoked from all specific principals, even if the specific principals already had that permission before it was made global.
- None of the global permissions can be revoked from a specific principal. The OMA DM ACL format does not handle exceptions, which must be enforced by the deletePermission and setPermission methods.

117.8.2 Ghost ACLs

The ACLs are fully maintained by the Dmt Admin service and enforced when the session has an associated principal. A plugin must be completely unaware of any ACLs. The Dmt Admin service must synchronize the ACLs with any change in the DMT that is made through its service interface. For example, if a node is deleted through the Dmt Admin service, it must also delete an associated ACL.

The DMT nodes, however, are mapped to plugins, and plugins can delete nodes outside the scope of the Dmt Admin service.

As an example, consider a configuration record which is mapped to a DMT node that has an ACL. If the configuration record is deleted using the Configuration Admin service, the data disappears, but the ACL entry in the Dmt Admin service remains. If the configuration dictionary is recreated with the same PID, it will get the old ACL, which is likely not the intended behavior.

This specification does not specify a solution to solve this problem. Suggestions to solve this problem are:

- Use a proprietary callback mechanism from the underlying representation to notify the Dmt Admin service to clean up the related ACLs.
- Implement the services on top of the DMT. For example, the Configuration Admin service could use a plugin that provides general data storage service.

117.9 Notifications

In certain cases it is necessary for some code on the device to alert a remote management server or to initiate a session; this process is called sending a notification or an *alert*. Some examples:

- A Plugin that must send the result of an asynchronous EXEC operation.
- Sending a request to the server to start a management session.
- Notifying the server of completion of a software update operation.

Notifications can be sent to a management server using the sendNotification(String,int,String,AlertItem[]) method on the Notification Service. This method is on the Notification Service and not on the session, because the session can already be closed when the need for an alert arises. If an alert is related to a session, the session can provide the required principal, even after it is closed.

The remote server is alerted with one or more AlertItem objects. The AlertItem class describes details of the alert. An alert code is an alert type identifier, usually requiring specifically formatted AlertItem objects.

The data syntax and semantics vary widely between various alerts, and so does the optionality of particular parameters of an alert item. If an item, such as source or type, is not defined, the corresponding getter method must return null.

The AlertItem class contains the following items. The value of these items must be defined in an alert definition:

- source - (String) The URI of a node that is related to this request. This parameter can be null.
- type - (String) The type of the item. For example, x-oma-application:syncml.samplealert in the Generic Alert example.
- mark - (String) Mark field of an alert. Contents depend on the alert type.
- data - (DmtData) The payload of the alert with its type.

An AlertItem object can be constructed with two different constructors:

- AlertItem(String,String,String,DmtData) - This method takes all the previously defined fields.
- AlertItem(String[],String,String,DmtData) - Same as previous but with a convenience parameter for a segmented URI.

The Notification Service provides the following method to send AlertItem objects to the management server:

- sendNotification(String,int,String,AlertItem[]) - Send the alert to the server that is associated with the session. The first argument is the name of the principal (identifying the remote management system) or null for implementation defined routing. The int argument is the *alert type*. The alert types are defined by *managed object types*. The third argument (String) can be used for the correlation id of a previous execute operation that triggered the alert. The AlertItem objects contain the data of the alert. The method will run asynchronously from the caller. The Notification Service must provide a reliable delivery method for these alerts. Alerts must therefore not be re-transmitted.

  When this method is called with null correlator, null or empty AlertItem array, and a 0 code as values, it should send a protocol specific notification that must initiate a new management session.

Implementers should base the routing on the session or server information provided as a parameter in the sendNotification(String,int,String,AlertItem[]) method. Routing might even be possible without any routing information if there is a well known remote server for the device.

If the request cannot be routed, the Alert Sender service must immediately throw a Dmt Exception with a code of ALERT_NOT_ROUTED. The caller should not attempt to retry the sending of the notification. It is the responsibility of the Notification Service to deliver the notification to the remote management system.

117.9.1 Routing Alerts

The Notification Service allows external parties to route alerts to their destination. This mechanism enables Protocol Adapters to receive any alerts for systems with which they can communicate.

Such a Protocol Adapter should register a Remote Alert Sender service. It should provide the following service property:

- *principals* - (String+) The array of principals to which this Remote Alert Sender service can route alerts. If this property is not registered, the Remote Alert Sender service will be treated as the default sender. The default alert sender is only used when a more specific alert sender cannot be found.

If multiple Remote Alert Sender services register for the same principals highest ranking service is taken as defined in the OSGi Core.

117.10 Exceptions

Most of the methods of this Dmt Admin service API throw Dmt Exceptions whenever an operation fails. The DmtException class contains numeric error codes which describe the cause of the error. Some of the error codes correspond to the codes described by the OMA DM spec, while some are introduced by the OSGi Working Group. The documentation of each method describes what codes could potentially be used for that method.

The fatality of the exception decides if a thrown Exception rolls back an atomic session or not. If the isFatal() method returns true, the Exception is fatal and the session must be rolled back.

All possible error codes are constants in the DmtException class.

117.11 Events

There are the following mechanisms to work with events when using the Dmt Admin service.

- *Event Admin service* - Standard asynchronous notifications
- *Dmt Event Listener service* - A white board model for listener. A registered DmtEventListener service can use service properties to filter the received events

In both cases events are delivered asynchronously and ordered per listener unless otherwise specified. Events to the DMT can occur because of modifications made in a session or they can occur because a Plugin changes its internal state and notifies the Dmt Admin service through the MountPoint interface.

Changes made through a session always start with a SESSION_OPENED event directly after the session is opened. This event must contain the properties defined in *Life Cycle Event Properties* on page 404.

If events originate from an atomic session then these events must be queued until the sessions is successfully committed, which can happen multiple times over the life time of a session. If the session is rolled back or runs into an error then none of the queued events must be sent.

When a session is closed, which can happen automatically when the session fails, then the SESSION_CLOSED event must be sent. This event must happen after any queued events. This closed event must contain the properties defined in *Life Cycle Event Properties* on page 404.

An event must only be sent when that type of event actually occurred.

117.11.1 Event Admin

Event Admin, when present, must be used to deliver the Dmt Admin events asynchronously. The event types are specified in Table 117.7 on page 402, the Topic column defines the Event Admin topic. The Table 117.10 on page 404 and Table 117.9 on page 404 define the Life Cycle and Session properties that must be passed as the event properties of Event Admin.

117.11.2 Dmt Event Listeners

To receive the Dmt Admin events it is necessary to register a Dmt Event Listener service. It is possible to filter the events by registering a combination of the service properties defined in the following table.

Table 117.6 *Service Properties for the Dmt Event Listener*

Service Property	Data Type	Default	Description
FILTER_EVENT	Integer	All Events	A bitmap of DmtEvent types: SESSION_OPENED, ADDED, COPIED, DELETED, RENAMED, REPLACED, and SESSION_CLOSED. A Dmt Event's type must occur in the bitmap to be delivered.
FILTER_PRINCIPAL	String+	Any node	Only deliver Dmt Events for which at least one of the given principals has the right to Get that node.
FILTER_SUBTREE	String+	Any node	This property defines a number of sub-trees by specifying the URI of the top nodes of these sub-trees. Only events that occur in one of the sub-trees must be delivered.

A Dmt Event must only be delivered to a Dmt Event Listener if the Bundle that registers the Dmt Event Listener service has the GET Dmt Permission for each of the nodes used in the nodes and newNodes properties as tested with the Bundle hasPermission method.

The Dmt Admin service must track Dmt Event Listener services and deliver matching events as long as a Dmt Event Listener service is registered. Any changes in the service properties must be expediently handled.

A Dmt Event Listener must implement the changeOccurred(DmtEvent) method. This method is called asynchronously from the actual event occurrence but each listener must receive the events in order.

Events are delivered with a DmtEvent object. This object provides access to the properties of the event. Some properties are available as methods others must be retrieved through the getProperty(String) method. The methods that provide property information are listed in the property tables, see Table 117.10 on page 404.

117.11.3 Atomic Sessions and Events

The intent of the events is that a listener can follow the modifications to the DMT from the events alone. However, from an efficiency point of view certain events should be coalesced to minimize the number of events that a listener need to handle. For this reason, the Dmt Admin service must coalesce events if possible.

Two consecutive events can be coalesced when they are of the same type. In that case the nodes and, if present, the newNodes of the second event can be concatenated with the first event and the timestamp must be derived from the first event. It is not necessary to remove duplicates from the nodes and newNodes. This guarantees that the order of the nodes is in the order of the events.

117.11.4 Event Types

This section describes the events that can be generated by the Dmt Admin service. Table 117.7 enumerates all the events and provides the name of the topic of Event Admin and the Dmt Event type for the listener model.

There are two kinds of events:

- *Life Cycle Events* - The events for session open and closed are the session events.
- *Session Events* - ADDED, DELETED, REPLACED, RENAMED, and COPIED.

Session and life cycle events have different properties.

Table 117.7 *Event Types*

Event	Topic	Dmt Event Type	Description
SESSION OPENED	org/osgi/service/dmt/DmtEvent/SESSION_OPENED	SESSION_OPENED	A new session was opened. The event must the properties defined in Table 117.9 on page 404.

Event	Topic	Dmt Event Type	Description
ADDED	org/osgi/service/dmt/DmtEvent/ADDED	ADDED	One or more nodes were added.
DELETED	org/osgi/service/dmt/DmtEvent/DELETED	DELETED	One or more existing nodes were deleted.
REPLACED	org/osgi/service/dmt/DmtEvent/REPLACED	REPLACED	Values of nodes were replaced.
RENAMED	org/osgi/service/dmt/DmtEvent/RENAMED	RENAMED	Existing nodes were renamed.
COPIED	org/osgi/service/dmt/DmtEvent/COPIED	COPIED	Existing nodes were copied. A copy operation does not trigger an ADDED event (in addition to the COPIED event), even though new node(s) are created. For efficiency reasons, recursive copy and delete operations must only generate a single COPIED and DELETED event for the root of the affected sub-tree.
SESSION CLOSED	org/osgi/service/dmt/DmtEvent/SESSION_CLOSED	SESSION_CLOSED	A session was closed either because it was closed explicitly or because there was an error detected. The event must the properties defined in Table 117.9 on page 404.

117.11.5 General Event Properties

The following properties must be available as the event properties in Event Admin service and the properties in the Dmt Event for Dmt Event Listener services.

Table 117.8 *General Event*

Property Name	Type	Dmt Event	Description
event.topics	String		Event topic, required by Event Admin but must also be present in the Dmt Events.
session.id	Integer	getSessionId()	A unique identifier for the session that triggered the event. This property has the same value as getSessionId() of the associated DMT session. If this event is generated outside a session then the session id must be -1, otherwise it must be >=1.
timestamp	Long		The time the event was started as defined by System.currentTimeMillis()
bundle	Bundle		The initiating Bundle, this is the bundle that caused the event. This is either the Bundle that opened the associated session or the Plugin's bundle when there is no session (i.e. the session id is -1).
bundle.signer	String+		The signer of the initiating Bundle
bundle.symbolicName	String		The Bundle Symbolic name of the initiating Bundle
bundle.version	Version		The Bundle version of the initiating Bundle.
bundle.id	Long		The Bundle Id of the initiating Bundle.

117.11.6 Session Event Properties

All session events must have the properties defined in the following table.

Table 117.9 Event Properties For Session Events

Property Name	Type	Dmt Session	Description
session.rooturi	String	getRootUri()	The root URI of the session that triggered the event.
session.principal	String	getPrincipal()	The principal of the session, or absent if no principal is associated with this session. In the latter case the method returns null.
session.locktype	Integer	getLockType()	The lock type of the session. The number is mapped as follows: • LOCK_TYPE_SHARED - 0 • LOCK_TYPE_EXCLUSIVE - 1 • LOCK_TYPE_ATOMIC - 2
session.timeout	Boolean		If the session timed out then this property must be set to true. If it did not time out this property must be false.
exception	Throwable		The name of the actual exception class if the session had a fatal exception.
exception.message	String		Must describe the exception if the session had a fatal exception.
exception.class	String		The name of the actual exception class if the session had a fatal exception.

117.11.7 Life Cycle Event Properties

All Life Cycle events must have the properties defined in the following table.

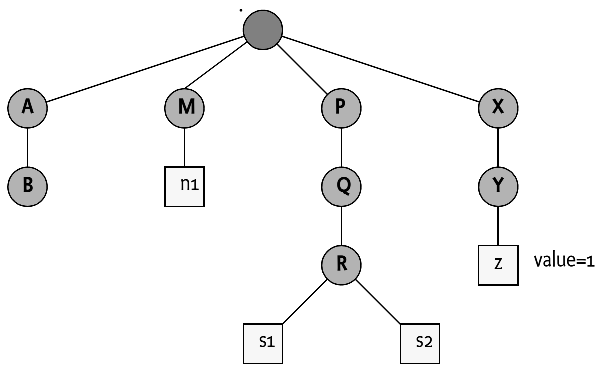
Table 117.10 Event Properties for Life Cycle Events

Property Name	Type	Dmt Event	Description
nodes	String[]	getNodes()	The absolute URIs of each affected node. This is the nodeUri parameter of the Dmt API methods. The order of the URIs in the array corresponds to the chronological order of the operations. In case of a recursive delete or copy, only the session root URI is present in the array.
newnodes	String[]	getNewNodes()	The absolute URIs of new renamed or copied nodes. Only the RENAMED and COPIED events have this property. The newnodes array runs parallel to the nodes array. In case of a rename, newnodes[i] must contains the new name of nodes[i], and in case of a copy, newnodes[i] is the URI to which nodes[i] was copied.

117.11.8 Example Event Delivery

The example in this section shows the change of a non-trivial tree and the events that these changes will cause.

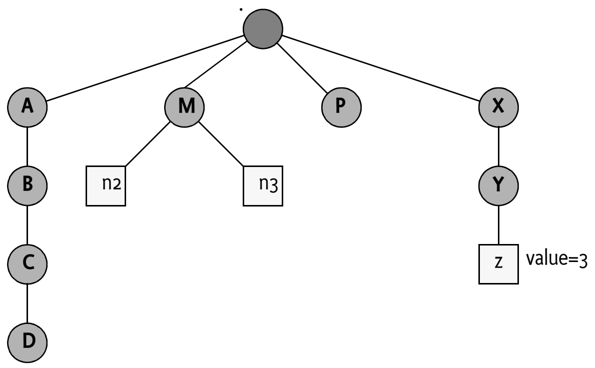
Figure 117.21 Example DMT before



For example, in a given session, when the DMT in Figure 117.21 is modified with the following operations:

- Open atomic session 42 on the root URI
- Add node ./A/B/C
- Add node ./A/B/C/D
- Rename ./M/n1 to./M/n2
- Copy ./M/n2 to ./M/n3
- Delete node ./P/Q
- Add node ./P/Q
- Delete node ./P/Q
- Replace ./X/Y/z with 3
- Commit
- Close

Figure 117.22 Example DMT after



When the Dmt Session is opened, the following event is published:

```
SESSION_OPENED {
        session.id = 42
        session.rooturi=.
        session.principal=null
        session.locktype=2
        timestamp=1313411544752
        bundle  =<Bundle>
        bundle.signer=[]
        bundle.symbolicname"com.acme.bundle"
        bundle.version=1.2.4711
        bundle.id=442
        ...
}
```

When the Dmt Session is closed (assuming it is atomic), the following events are published:

```
ADDED {
        nodes = [./A/B/C, ./A/B/C/D ]       # note the coalescing
        session.id = 42
        ...
}
RENAMED {
        nodes = [ ./M/n1 ]
        newnodes = [ ./M/n2 ]
        session.id = 42
        ...
}
COPIED {
        nodes = [ ./M/n2 ]
        newnodes = [ ./M/n3 ]
        session.id = 42
        ...
}
DELETED {
        nodes = [ ./P/Q ]
        session.id = 42
        ...
}
ADDED {
        nodes = [ ./P/Q ]
        session.id = 42
        ...
}
DELETED {
        nodes = [ ./P/Q ]
        session.id = 42
        ...
}
REPLACED {
        nodes = [ ./X/Y/z ]
        session.id = 42
        ...
}
SESSION_CLOSED {
        session.id = 42
        session.rooturi=.
        session.principal=null
        session.locktype=2
        ...
}
```

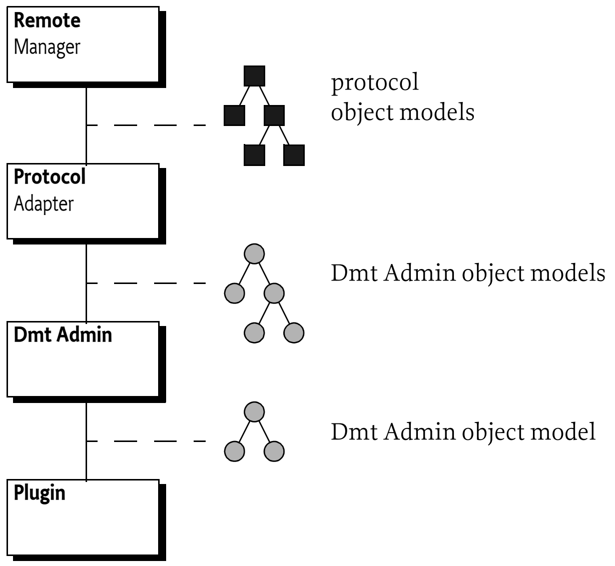
117.12 OSGi Object Modeling

117.12.1 Object Models

Management protocols define only half the picture; the object models associated with a particular protocol are the other half. Object models are always closely associated with a remote management protocol since they are based on the data types and actions that are defined in the protocol. Even small differences between the data types of a protocol and its differences make accurate mapping between protocols virtually impossible. It is therefore necessary to make the distinction between *native* and *foreign* protocols for an object model.

A native protocol for an object model originates from the same specification organization. For example, OMA DM consists of a protocol based on SyncML and a number of object models that define the structure and behavior of the nodes of the DMT. The FOMA specification defines an OMA DM native object model, it defines how firmware management is done. This is depicted in Figure 117.23.

Figure 117.23 Device Management Architecture



If an object implements a standardized data model it must be visible through its *native* Protocol Adapter, that is the Protocol Adapter that belongs to the object model's standard. For example, an ExecutionUnit node defined in UPnP Device Management could be implemented as a bundle, exposed through a Data Plugin for the Dmt Admin service, and then translated by its native UPnP Protocol Adapter.

If an object is present in the Dmt Admin service it is also available to *foreign* Protocol Adapters. A foreign Protocol Adapter is any Protocol Adapter except its native Protocol Adapter. For example, the Broadband Forum's ExecutionUnit could be browsed on the foreign OMA DM protocol.

In a foreign Protocol Adapter the object model should be *browsable* but it would not map to one of its native object models. Browsable means that the information is available to the Protocol Adapter's remote manager but not recognized as a standard model for the manager. Browse can include, potentially limited, manipulation.

In a native Protocol Adapter it is paramount that the mapping from the DMT to the native object is fully correct. It is the purpose of this part of the Dmt Admin service specification to allow the native Protocol Adapter to map the intentions of the Plugin without requiring knowledge of the specific native object model. That is, a TR-069 Plugin implementing a WAN interface must be available over the TR-069 protocol without the Protocol Adapter having explicit knowledge about the WAN interfaces object models from Broadband Forum.

Therefore, the following use cases are recognized:

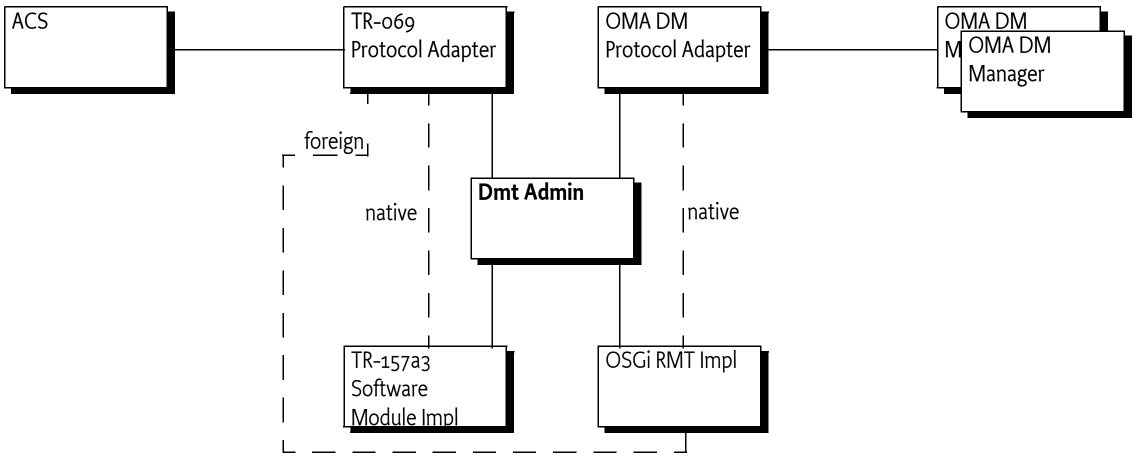
- *Foreign Mapping* - Foreign mapping can is best-effort as there is no object model to follow. Each Protocol Adapter must define how the Dmt Admin model is mapped for this browse mode.
- *Native Mapping* - Native mapping must be 100% correct. As it is impossible automatically map DMTs to arbitrary protocols this specification provides the concept of a mapping model that allows a Plugin to instruct its native Protocol Adapter using Meta Nodes.

117.12.2 Protocol Mapping

The OSGi Working Group specifies an Execution Environment that can be used as a basis for residential gateways, mobiles, or other devices. This raises the issue how to expose the manageability of an OSGi device and the *objects*, the units of manageability, that are implemented through Plugins. Ideally, an object should be able to expose its management interface once and then Protocol Adapters convert the management interface to specific device management stacks. For example, an object can be exposed through the Dmt Admin service where then a TR-069 Protocol Adapter maps the DMT to the TR-069 Remote Procedure Calls (RPC).

Figure 117.24 shows an example of a TR-069 Protocol Adapter and an OMA DM Protocol Adapter. The TR-069 Protocol Adapter should be able to map native TR-069 objects in the DMT (the `Software Modules Impl` in the figure) to Broadband Forum's object models. It should also be able to browse the foreign DMT and other objects that are not defined in Broadband forum but can be accessed with the TR-069 RPCs.

Figure 117.24 Implementing & Browsing



A *Protocol Mapping* is a document that describes the default mapping and the native mechanism for exact mapping.

The following sections specify how Plugins must implement an object model that is exposed through the Dmt Admin service. This model is limited from the full Dmt Admin service capabilities so that for each protocol it is possible to specify a default mapping for browsing as well as a mechanism to ensure that special conversion requirements can be communicated from a Plugin to its native Protocol Adapter.

117.12.3 Hierarchy

The Dmt Admin model provides an hierarchy of *nodes*. Each node has a *type* that is reflected by its Meta Node. A node is addressed with a URI. The flexibility of the Dmt Admin service allows a large number of constructs, for example, the name of the node can be used as a *value*, a feature that some management standards support. To simplify mapping to foreign Protocol Adapters, some of the fundamental constructs have been defined in the following sections.

117.12.4 General Restriction Guidelines

The Dmt Admin service provides a very rich tool to model complex object structures. Many choices can be made that would make it very hard to browse DMTs on non-OMA DM protocols or make the DMT hard to use through the Dmt Admin service. As Plugins can always signal special case handling to their native Protocol Adapter, any object model design should strive to be easy to use for the developers and managers. Therefore, this section provides a number of guidelines for the design of such object models that will improve the browsing experience for many Protocol Adapters.

- *Reading of a node must not change the state of a device* - Management systems must be able to browse a tree without causing any side effects. If reading modified the DMT, a management system would have no way to warn the user that the system is modified. There are a number of technical reasons as well (race conditions, security holes, and eventing) but the most important reason is the browsability of the device.
- *No use of recursive structures* - The Dmt Admin service provides a very rich tree model that has no problem with recursion. However, this does not have to be true for other models. To increase the changes that a model is browsable on another device it is strongly recommended to prevent recursive models. For example, TR-069 cannot handle recursive models.
- *Only a single format per meta node* - Handling different types in different nodes simplifies the data conversion for both foreign and native protocols. Having a single choice from the Meta Node makes the conversion straightforward and does not require guessing.
- *All nodes must provide a Meta Node* - Conversion without a Meta Node makes the conversion very hard since object model schemas are often not available in the Protocol Adapter.
- *Naming* - Structured node members must have names only consisting of [a-zA-Z0-9] and must always start with a character [a-zA-z]. Member names must be different regardless of the case, that is Abc and ABC must not both be members of the same structured node. The reason for this restriction is that it makes it more likely that the chosen names are compatible with the supported protocols and do not require escaping.
- *Typing* - Restrict the used formats to formats that maximize both the interoperability as the ease of use for Java developers. The following type are widely supported and are easy to use from Java:
  - FORMAT_STRING
  - FORMAT_BOOLEAN
  - FORMAT_INTEGER
  - FORMAT_LONG
  - FORMAT_FLOAT
  - FORMAT_DATE_TIME
  - FORMAT_BINARY

117.12.5 DDF

The Data Description Format is part of OMA DM; it provides a description language for the object model. The following table provides an example of the Data Description Format as used in the OSGi specifications.

Name	Actions	Type	Card.	S	Description
FaultType	Get	integer	1	P	...

The columns have the following meanings:

- *Name* - The name of the node

- *Actions* - The set of actions that can be executed on the node, see *Operations* on page 380.
- *Type* - The type of the node. All lower case are primitives, a name starting with an upper case is an interior node type. MAP, LIST, and SCAFFOLD are the special types. The NODE type is like an ANY type. Other type names are then further specified in the document. See *Types* on page 410.
- *Cardinality* - The number of occurrences of the node, see *Cardinality* on page 381.
- *Scope* - The scope of the node, see *Scope* on page 380.
- *Description* - A description of the node.

117.12.6 Types

Each node is considered to have a *type*. The Dmt Admin service has a number of constructs that have typing like behavior. There are therefore the following *kind* of types:

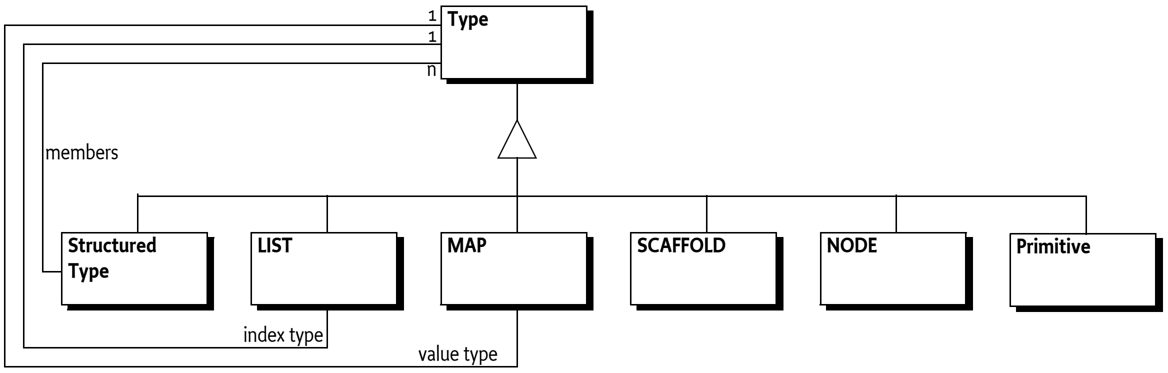
- *Primitives* - Primitives are data types like integers and strings; they include all the Dmt Admin data formats. See *Primitives* on page 411. Primitive type names are always lower case to distinguish them from the interior node type names.
- *Structured Types* - A structured type types a structured node. See *Structured Nodes* on page 411. A structured type has a type name that starts with an uppercase. Object models generally consist of defining these types.
- NODE - A general unqualified Dmt Admin node.
- LIST - A node that represents a homogeneous collection of child nodes; the name of the child nodes is the index in the collection. See *LIST Nodes* on page 411.
- MAP - A node that represents a mapping from a key, the name of the child node, and a value, the value of the child node. All values have the same type. See *MAP Nodes* on page 413.
- SCAFFOLD - A node provided by the Dmt Admin service or a Parent Plugin to make it possible to discover a DMT, see *Scaffold Nodes* on page 385.

Nodes are treated as if there is a single type system. However, the Dmt Admin type system has the following mechanisms to type a node:

- *Format* - The Dmt Admin primitive types used for leaf nodes, as defined on Dmt Data.
- *MIME* - A MIME type on a leaf node which is available through getNodeType(String).
- *DDF Document URI* - A Data Description Format URI that provides a type name for an interior node. The URI provides a similar role as the MIME type for the leaf node and is also available through getNodeType(String).

The Dmt Admin service provides the MIME type for leaf nodes and the DDF Document URI for interior nodes through the getNodeType(String) method. As both are strings they can both be used as type identifiers. The different types are depicted in Figure 117.25.

Figure 117.25 Type inheritance and structure



117.12.7 Primitives

A primitive is a value stored in a leaf node. In the Dmt Admin service, the type of the primitive is called the *format*. The Dmt Admin service supports a large number of types that have semantic overlap. A Protocol Mapping must provide a unique mapping from each Dmt Admin format to the corresponding protocol type and provide conversion from a protocol type to the corresponding Dmt Admin types defined in a Meta Node.

Primitives are documented in OSGi object models with a lower case name that is the last part of their format definition. For example, for FORMAT_STRING the DDF type name is string. A primitive DDF for an integer leaf node therefore looks like:

Name	Act	Type	Card.	S	Description
FaultType	Get	integer	1	P	...

117.12.8 Structured Nodes

A *structured node* is like a struct in C or a class in an object oriented languages. A structured node is an interior node with a set of members (child nodes) with fixed names, it is never possible to add or remove such members dynamically. The meaning of each named node and its type is usually defined in a management specification. For example, a node representing the OSGi Bundle could have a BundleId child-node that maps to the getBundleId() method on the Bundle interface.

It is an error to add or delete members to a Structured node, this must be reflected in the corresponding Meta Node, that is, Structured nodes must never have the Add or Delete action.

A structured node is defined in a *structured type* to allow the reuse of the same information in different places in an object model. A structured type defines the members and their behaviors. A structured type can be referred by its name. The name of the type is often, but not required, the name of the member.

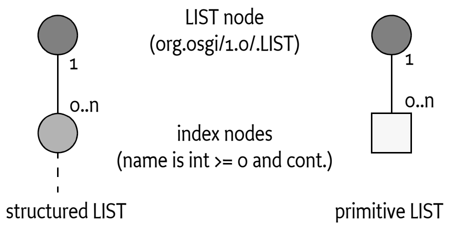
For example, a Unit structured type could look like:

Name	Act	Type	Card.	S	Description
Id	Get	long	1	P	...
URL	Get Set	string	1	P	...
Name	Get	string	1	P	...
Certificate	Get	LIST	1	P	
[index]	Get	Certificate	1	D	Note the use of a structured type.

117.12.9 LIST Nodes

A LIST node is an interior node representing a *collection* of elements. The elements are stored in the child nodes of the LIST node, they are called the *index nodes*. All index nodes must have the same type. The names of the index nodes are synthesized and represent the index of the index node. The first node is always named 0 and the sibling is 1, 2, etc. The sequence must be continuous and must have no missing indexes. A node name is always a string, it is therefore the responsibility of the plugin to provide the proper names. The index is assumed to be a signed positive integer limiting the LIST nodes size to Integer.MAX_VALUE elements.

Figure 117.26 LIST Nodes



Index nodes should only be used for types where the value of the index node is the identity. For example, a network interface has an identity; a manager will expect that a node representing such as a network interface node will always have the same URI even if other interfaces are added and deleted. Since LIST nodes renumber the index node names when an element is deleted or added, the URI would fail if a network interface was added or removed. If such a case, a MAP node should be used, see *MAP Nodes* on page 413, as they allow the key to be managed by the remote manager.

LIST nodes can be mutable if the Meta Node of its index nodes support the Add or Delete action. A LIST node is modeled after a java.util.List that can automatically accommodate new elements. Get and Replace operations use the node name to index in this list.

To rearrange the list the local manager can Add and Delete nodes or rename them as it sees fit. At any moment in time the underlying implementation must maintain a list that runs from 0 to max(index) (inclusive), where index is the name of the LIST child nodes. Inserting a node requires renaming all subsequent nodes. Any missing indexes must automatically be provided by the plugin when the child node names are retrieved.

For example, a LIST node named L contains the following nodes:

```
L/0     A
L/1     B
L/2     C
```

To insert a node after B, L/2 must be renamed to L/3. This will automatically extend the LIST node to 4 elements. That is, even though L/2 is renamed, the implementation must automatically provide a new L/2 node. The value of this node depends on the underlying implementation. The value of the list will therefore then be: [A,B,?,C]. If node 1 is deleted, then the list will be [A,?,C]. If a node L/5 is added then the list will be [A,?,C,?,?,?]. It is usually easiest to use the LIST node as a complex value, this is discussed in the next section.

117.12.9.1 Complex Collections

An implementation of a LIST node must support a complex node value if its members are primitive; the interior node must then have a value of a Java object implementing the Collection interface from java.util. The elements in this map must be converted according to the following table.

Table 117.11 Conversion for Collections

Format	Associated Java Type
FORMAT_STRING	String
FORMAT_BOOLEAN	Boolean
FORMAT_INTEGER	Integer
FORMAT_LONG	Long
FORMAT_FLOAT	Float
FORMAT_DATE_TIME	Date
FORMAT_BINARY	byte[]

Alternatively, the Collection may contain Dmt Data objects but the collection must be homogeneous. The collection must always be a copy and changes made to the collection must not affect the DMT.

For example, a LIST type for a list of URIs could look like:

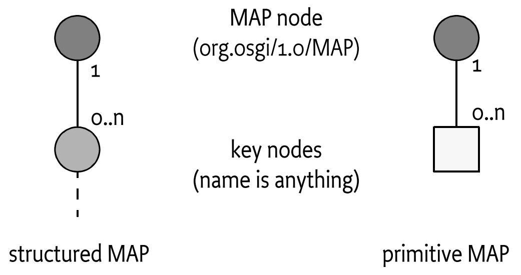
Name	Act	Type	Card.	S	Description
URIs	Get	LIST	1	P	A List of URIs
[index]	Get Set Add Del	string	0..n	D	A primitive index node

Replacing a complex value will generate a single EVENT_TOPIC_REPLACED event for the LIST node.

117.12.10 MAP Nodes

A MAP node represents a mapping from a *key* to a *value*. The key is the name of the node and the value is the node's value. A MAP node performs the same functions as a Java Map. See Figure 117.27.

Figure 117.27 *MAP Nodes*



A MAP node has *key nodes* as children. A key node is an association between the name of the key node (which is the key) and the value of the key node. Key nodes are depicted with [<type>], where the <type> indicates the type used for the string name. For example, a long type will have node names that can be converted to a long. A key type must always be one of the primitive types. For example, a list of Bundle locations can be handled with a MAP with [string] key nodes that have a value type of string. Since the key is used in URIs it must always be escaped, see *The DMT Addressing URI* on page 371.

For example:

Name	Act	Type	Card.	S	Description
Location	Get	MAP	1	P	A MAP of location where the index node is the Bundle Id.
[long]	Get Set Add Del	string	0..n	D	Name is the Bundle Id and the value is the location.

117.12.10.1 Complex Value

An implementation of a MAP node must support an interior node value if its child nodes are primitive; the interior node must then be associated with a Java object implementing the Map interface from java.util. The values in this Map must homogeneous and be converted according to Table 117.11 or the given values must of type DmtData. The Map object must a copy and does not track changes in the DMT or vice-versa.

Replacing a complex value will generate a single EVENT_TOPIC_REPLACED event for that node.

117.12.11 Instance Id

Some protocols cannot handle arbitrary names in the access URI, they need a well defined *instance id* to index in a table or put severe restrictions on the node name's character set, length, or other aspects. For example, TR-069 can access an object with the following URI:

```
Device.VOIP.12.Name
```

The more natural model for the DMT is to use:

```
Device.VOIP.<Name>...
```

To provide assistance to these protocols this section defines a mechanism that can be used by Protocol Adapters to simplify access.

An Object Model can define a child node `InstanceId`. The `InstanceId` node, if present, holds a long value that has the following qualities:

- Its value must be between 1 and `Long.MAX_VALUE`.
- No other index/key node on the same level must have the same value for the `InstanceId` node
- The value must be persistent between sessions and restarts of the plugin
- A value must not be reused when a node is deleted until the number space is exhausted

Protocol Adapters can use this information to provide alternative access paths for the DMT.

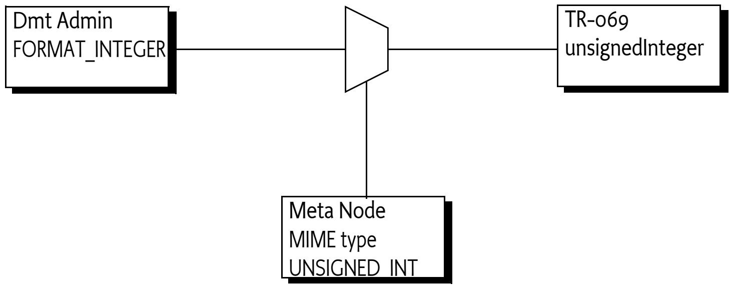
117.12.12 Conversions

Each Protocol Mapping document should define a default conversion from the Dmt Admin data formats to the protocol types and vice versa, including the `LIST` and `MAP` nodes. However, this default mapping is likely to be too constraining in real world models since different protocols support different data types and a 1:1 mapping is likely to be impossible.

For this reason, the Protocol Mapping document should define a number of protocol specific MIME types for each unique data type that they support. A Data Plugin can associate such a MIME type with a node. The Protocol Adapter can then look for this MIME type. If none of the Protocol Adapter specific MIME types are available in a node the default conversion is used.

For example, in the TR-069 Protocol Adapter specification there is a MIME type for each TR-069 data type. If for a given leaf node the Meta Node's type specifies `TR069_MIME_UNSIGNED_INT` and the node specifies the format `FORMAT_INTEGER` then the Protocol Adapter must convert the integer to an unsigned integer and encode the value as such in the response message. The Protocol Adapter there does not have to have specific knowledge of the object model, the Plugin drives the Protocol Adapter by providing the protocol specific MIME types on the leaf node Meta Nodes. This model is depicted in Figure 117.28.

Figure 117.28 *Conversions*



Since a Meta Node can contain multiple MIME types, there is no restrictions on the number of Protocol Adapters; a Plugin can specify the MIME types of multiple Protocol Adapters.

117.12.13 Extensions

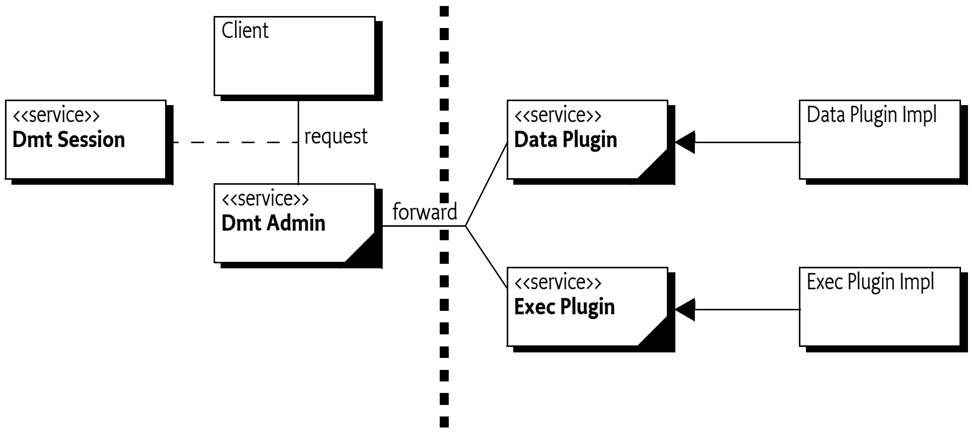
All interior nodes in this specification can have a node named Ext. These nodes are the *extension* nodes. If an implementation needs to expose additional details about an interior node then they should expose these extensions under the corresponding Ext node. To reduce name conflicts, it is recommended to group together implementation specific extensions under a unique name, recommended is to use the reverse domain name. For example, the following DDF defines an Ext node with extensions for the ACME provider.

Name	Act	Type	Card.	S	Description
Framework	Get	Framework	1	P	...
Ext	Get		1	P	Extension node
com.acme	Get	AcmeFrameworkExt	1	P	The node for the ACME extensions
Transactional	Get	boolean	1	P	...

117.13 Security

A key aspect of the Dmt Admin service model is the separation from DMT clients and plugins. The Dmt Admin service receives all the operation requests and, after verification of authority, forwards the requests to the plugins.

Figure 117.29 Separation of clients and plugins



This architecture makes it straightforward to use the OSGi security architecture to protect the different actors.

117.13.1 Principals

The caller of the getSession(String,String,int) method must have the Dmt Principal Permission with a target that matches the given principal. This Dmt Principal Permission is used to enforce that only trusted entities can act on behalf of remote managers.

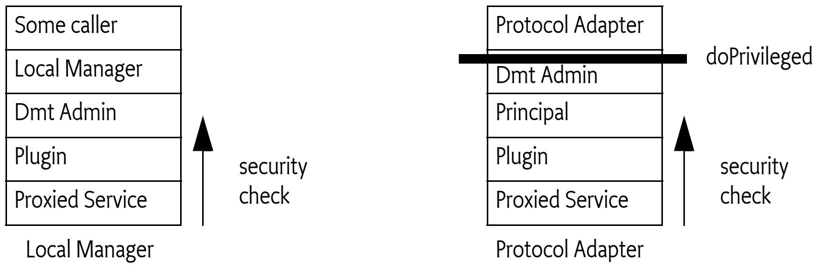
The Dmt Admin service must verify that all operations from a session with a principal can be executed on the given nodes using the available ACLs.

The other two forms of the getSession method are meant for local management applications where no principal is available. No special permission is defined to restrict the usage of these methods. The callers that want to execute device management commands, however, need to have the appropriate Dmt Permissions.

117.13.2 Operational Permissions

The operational security of a Local Manager and a remote manager is distinctly different. The distinction is made on the principal. Protocol Adapters should use the getSession method that takes an authenticated principal. Local Managers should not specify a principal.

Figure 117.30 Access control context, for Local Manager and Protocol Adapter operation



117.13.3 Protocol Adapters

A Protocol Adapter must provide a principal to the Dmt Admin service when it gets a session. It must use the getSession(String,String,int) method. The Protocol Adapter must have Dmt Principal Permission for the given principal. The Dmt Admin service must then use this principal to determine the *security scope* of the given principal. This security scope is a set of permissions. How these permissions are found is not defined in this specification; they are usually in the management tree of a device. For example, the Mobile Specification stores these under the $/Policy/Java/DmtPrincipalPermission sub-tree.

Additionally, a Dmt Session with a principal implies that the Dmt Admin service must verify the ACLs on the node for all operations.

Any operation that is requested by a Protocol Adapter must be executed in a doPrivileged block that takes the principal's security scope. The doPrivileged block effectively hides the permissions of the Protocol Adapter; all operations must be performed under the security scope of the principal.

The security check for a Protocol Adapter is therefore as follows:

- The operation method calls doPrivileged with the security scope of the principal.
- The operation is forwarded to the appropriate plugin. The underlying service must perform its normal security checks. For example, the Configuration Admin service must check for the appropriate Configuration Permission.

The Access Control context is shown in Figure 117.30 within the Protocol Adapter column.

This principal-based security model allows for minimal permissions on the Protocol Adapter, because the Dmt Admin service performs a doPrivileged on behalf of the principal, inserting the permissions for the principal on the call stack. This model does not guard against malicious Protocol Adapters, though the Protocol Adapter must have the appropriate Dmt Principal Permission.

The Protocol Adapter is responsible for the authentication of the principal. The Dmt Admin service must trust that the Protocol Adapter has correctly verified the identity of the other party. This specification does not address the type of authentication mechanisms that can be used. Once it has permission to use that principal, it can use any DMT command that is permitted for that principal at any time.

117.13.4 Local Manager

A Local Manager does not specify a principal. Security checks are therefore performed against the security scope of the Local Manager bundle, as shown in Figure 117.30 with the Local Manager stack. An operation is checked only with a Dmt Permission for the given node URI and operation. A thrown Security Exception must be passed unmodified to the caller of the operation method. The Dmt Admin service must not check the ACLs when no principal is set.

A Local Manager, and all its callers, must therefore have sufficient permission to handle the DMT operations as well as the permissions required by the plugins when they proxy other services (which is likely an extensive set of Permissions).

117.13.5 Plugin Security

Plugins are required to hold the maximum security scope for any services they proxy. For example, the plugin that manages the Configuration Admin service must have ConfigurationPermission("*","*") to be effective.

Plugins should not make doPrivileged calls, but should use the caller's context on the stack for permission checks.

117.13.6 Events and Permissions

Dmt Event Listener services must have the appropriate Dmt Permission to receive the event since this must be verified with the hasPermission() method on Bundle.

The Dmt Event Listener services registered with a FILTER_PRINCIPAL service property requires Dmt Principal Permission for the given principal. In this case, the principal must have Get access to see the nodes for the event. Any nodes that the listener does not have access to must be removed from the event.

Plugins are not required to have access to the Event Admin service. If they send an event through the MountPoint interface then the Dmt Admin service must use a doPrivileged block to send the event to the Event Admin service.

117.13.7 Dmt Principal Permission

Execution of the getSession methods of the Dmt Admin service featuring an explicit principal name is guarded by the Dmt Principal Permission. This permission must be granted only to Protocol Adapters that open Dmt Sessions on behalf of remote management servers.

The DmtPrincipalPermission class does not have defined actions; it must always be created with a * to allow future extensions. The target is the principal name. A wildcard character is allowed at the end of the string to match a prefix.

Example:

```
new DmtPrincipalPermission("com.acme.dep*","*" )
```

117.13.8 Dmt Permission

The Dmt Permission controls access to management objects in the DMT. It is intended to control only the *local* access to the DMT. The Dmt Permission target string identifies the target node's URI (absolute path is required, starting with the './' prefix) and the action field lists the management commands that are permitted on the node.

The URI can end in a wildcard character * to indicate it is a prefix that must be matched. This comparison is string based so that node boundaries can be ignored.

The following actions are defined:

- ADD
- DELETE
- EXEC
- GET
- REPLACE

For example, the following code creates a Dmt Permission for a bundle to add and replace nodes in any URI that starts with ./D.

```
new DmtPermission("./D*", "Add,Replace")
```

This permission must imply the following permission:

```
new DmtPermission("./Dev/Operator/Name", "Replace")
```

117.13.9 Alert Permission

The Alert Permission permits the holder of this permission to send a notification to a specific *target principal*. The target is identical to *Dmt Principal Permission* on page 417. No actions are defined for Alert Permission.

117.13.10 Security Summary

117.13.10.1 Dmt Admin Service and Notification Service

The Dmt Admin service is likely to require All Permission. This requirement is caused by the plug-in model. Any permission required by any of the plugins must be granted to the Dmt Admin service. This set of permissions is large and hard to define. The following list shows the minimum permissions required if the plugin permissions are left out.

ServicePermission	..DmtAdmin	REGISTER
ServicePermission	..NotificationService	REGISTER
ServicePermission	..DataPlugin	GET
ServicePermission	..ExecPlugin	GET
ServicePermission	..EventAdmin	GET
ServicePermission	..RemoteAlertSender	GET
ServicePermission	..DmtEventListener	GET
DmtPermission	*	*
DmtPrincipalPermission	*	*
PackagePermission	org.osgi.service.dmt	EXPORTONLY
PackagePermission	org.osgi.service.dmt.spi	EXPORTONLY
PackagePermission	org.osgi.service.dmt.notification	EXPORTONLY
PackagePermission	org.osgi.service.dmt.notification.spi	EXPORTONLY
PackagePermission	org.osgi.service.dmt.registry	EXPORTONLY
PackagePermission	org.osgi.service.dmt.security	EXPORTONLY

117.13.10.2 Dmt Event Listener Service

ServicePermission	..DmtEventListener	REGISTER
PackagePermission	org.osgi.service.dmt	IMPORT

Dmt Event Listeners must have the appropriate DmtPermission to see the nodes in the events. If they are registered with a principal then they also need DmtPrincipalPermission for the given principals.

117.13.10.3 Data and Exec Plugin

ServicePermission	..NotificationService	GET
ServicePermission	..DataPlugin	REGISTER
ServicePermission	..ExecPlugin	REGISTER
PackagePermission	org.osgi.service.dmt	IMPORT
PackagePermission	org.osgi.service.dmt.notification	IMPORT
PackagePermission	org.osgi.service.dmt.spi	IMPORT
PackagePermission	org.osgi.service.dmt.security	IMPORT

The plugin is also required to have any permissions to call its underlying services.

117.13.10.4 Local Manager

```
ServicePermission   ..DmtAdmin                          GET
PackagePermission   org.osgi.service.dmt                IMPORT
PackagePermission   org.osgi.service.dmt.security       IMPORT
DmtPermission       <scope>                             ...
```

Additionally, the Local Manager requires all permissions that are needed by the plugins it addresses.

117.13.10.5 Protocol Adapter

The Protocol Adapter only requires Dmt Principal Permission for the instances that it is permitted to manage. The other permissions are taken from the security scope of the principal.

```
ServicePermission   ..DmtAdmin                          GET
ServicePermission   ..RemoteAlertSender                 REGISTER
PackagePermission   org.osgi.service.dmt                IMPORT
PackagePermission   org.osgi.service.dmt.notification.spi IMPORT
PackagePermission   org.osgi.service.dmt.notification   IMPORT
DmtPrincipalPermission <scope>
```

117.14 org.osgi.service.dmt

Device Management Tree Package Version 2.0.

This package contains the public API for the Device Management Tree manipulations. Permission classes are provided by the org.osgi.service.dmt.security package, and DMT plugin interfaces can be found in the org.osgi.service.dmt.spi package. Asynchronous notifications to remote management servers can be sent using the interfaces in the org.osgi.service.dmt.notification package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt; version="[2.0,2.1)"
```

117.14.1 Summary

- Acl - Acl is an immutable class representing structured access to DMT ACLs.
- DmtAdmin - An interface providing methods to open sessions and register listeners.
- DmtConstants - Defines standard names for DmtAdmin.
- DmtData - An immutable data structure representing the contents of a leaf or interior node.
- DmtEvent - Event class storing the details of a change in the tree.
- DmtEventListener - Registered implementations of this class are notified via DmtEvent objects about important changes in the tree.
- DmtException - Checked exception received when a DMT operation fails.
- DmtIllegalStateException - Unchecked illegal state exception.
- DmtSession - DmtSession provides concurrent access to the DMT.
- MetaNode - The MetaNode contains meta data as standardized by OMA DM but extends it (without breaking the compatibility) to provide for better DMT data quality in an environment where many software components manipulate this data.

- Uri - This class contains static utility methods to manipulate DMT URIs.

117.14.2 public final class Acl

Acl is an immutable class representing structured access to DMT ACLs. Under OMA DM the ACLs are defined as strings with an internal syntax.

The methods of this class taking a principal as parameter accept remote server IDs (as passed to DmtAdmin.getSession), as well as " * " indicating any principal.

The syntax for valid remote server IDs:

<*server-identifier*> ::= All printable characters except '=', '&', '*', '+' or white-space characters.

117.14.2.1 public static final int ADD = 2

Principals holding this permission can issue ADD commands on the node having this ACL.

117.14.2.2 public static final int ALL_PERMISSION = 31

Principals holding this permission can issue any command on the node having this ACL. This permission is the logical OR of ADD, DELETE, EXEC, GET and REPLACE permissions.

117.14.2.3 public static final int DELETE = 8

Principals holding this permission can issue DELETE commands on the node having this ACL.

117.14.2.4 public static final int EXEC = 16

Principals holding this permission can issue EXEC commands on the node having this ACL.

117.14.2.5 public static final int GET = 1

Principals holding this permission can issue GET command on the node having this ACL.

117.14.2.6 public static final int REPLACE = 4

Principals holding this permission can issue REPLACE commands on the node having this ACL.

117.14.2.7 public Acl(String acl)

acl The string representation of the ACL as defined in OMA DM. If null or empty then it represents an empty list of principals with no permissions.

□ Create an instance of the ACL from its canonical string representation.

Throws IllegalArgumentException– if acl is not a valid OMA DM ACL string

117.14.2.8 public Acl(String[] principals, int[] permissions)

principals The array of principals

permissions The array of permissions

□ Creates an instance with a specified list of principals and the permissions they hold. The two arrays run in parallel, that is principals[i] will hold permissions[i] in the ACL.

A principal name may not appear multiple times in the 'principals' argument. If the "*" principal appears in the array, the corresponding permissions will be granted to all principals (regardless of whether they appear in the array or not).

Throws IllegalArgumentException– if the length of the two arrays are not the same, if any array element is invalid, or if a principal appears multiple times in the principals array

117.14.2.9 public synchronized Acl addPermission(String principal, int permissions)

principal The entity to which permissions should be granted, or "*" to grant permissions to all principals.

permissions The permissions to be given. The parameter can be a logical or of more permission constants defined in this class.

□ Create a new Acl instance from this Acl with the given permission added for the given principal. The already existing permissions of the principal are not affected.

Returns a new Acl instance

Throws IllegalArgumentException– if principal is not a valid principal name or if permissions is not a valid combination of the permission constants defined in this class

117.14.2.10 public synchronized Acl deletePermission(String principal, int permissions)

principal The entity from which permissions should be revoked, or "*" to revoke permissions from all principals.

permissions The permissions to be revoked. The parameter can be a logical or of more permission constants defined in this class.

□ Create a new Acl instance from this Acl with the given permission revoked from the given principal. Other permissions of the principal are not affected.

Note, that it is not valid to revoke a permission from a specific principal if that permission is granted globally to all principals.

Returns a new Acl instance

Throws IllegalArgumentException– if principal is not a valid principal name, if permissions is not a valid combination of the permission constants defined in this class, or if a globally granted permission would have been revoked from a specific principal

117.14.2.11 public boolean equals(Object obj)

obj the object to compare with this Acl instance

□ Checks whether the given object is equal to this Acl instance. Two Acl instances are equal if they allow the same set of permissions for the same set of principals.

Returns true if the parameter represents the same ACL as this instance

117.14.2.12 public synchronized int getPermissions(String principal)

principal The entity whose permissions to query, or "*" to query the permissions that are granted globally, to all principals

□ Get the permissions associated to a given principal.

Returns The permissions of the given principal. The returned int is a bitmask of the permission constants defined in this class

Throws IllegalArgumentException– if principal is not a valid principal name

117.14.2.13 public String[] getPrincipals()

□ Get the list of principals who have any kind of permissions on this node. The list only includes those principals that have been explicitly assigned permissions (so "*" is never returned), globally set permissions naturally apply to all other principals as well.

Returns The array of principals having permissions on this node.

117.14.2.14 public int hashCode()

□ Returns the hash code for this ACL instance. If two Acl instances are equal according to the equals(Object) method, then calling this method on each of them must produce the same integer result.

Returns hash code for this ACL

117.14.2.15 public synchronized boolean isPermitted(String principal, int permissions)

principal The entity to check, or "*" to check whether the given permissions are granted to all principals globally

permissions The permissions to check

□ Check whether the given permissions are granted to a certain principal. The requested permissions are specified as a bitfield, for example (Acl.ADD | Acl.DELETE | Acl.GET).

Returns true if the principal holds all the given permissions

Throws IllegalArgumentException– if principal is not a valid principal name or if permissions is not a valid combination of the permission constants defined in this class

117.14.2.16 public synchronized Acl setPermission(String principal, int permissions)

principal The entity to which permissions should be granted, or "*" to globally grant permissions to all principals.

permissions The set of permissions to be given. The parameter is a bitmask of the permission constants defined in this class.

□ Create a new Acl instance from this Acl where all permissions for the given principal are overwritten with the given permissions.

Note, that when changing the permissions of a specific principal, it is not allowed to specify a set of permissions stricter than the global set of permissions (that apply to all principals).

Returns a new Acl instance

Throws IllegalArgumentException– if principal is not a valid principal name, if permissions is not a valid combination of the permission constants defined in this class, or if a globally granted permission would have been revoked from a specific principal

117.14.2.17 public synchronized String toString()

□ Give the canonical string representation of this ACL. The operations are in the following order: {Add, Delete, Exec, Get, Replace}, principal names are sorted alphabetically.

Returns The string representation as defined in OMA DM.

117.14.3 public interface DmtAdmin

An interface providing methods to open sessions and register listeners. The implementation of DmtAdmin should register itself in the OSGi service registry as a service. DmtAdmin is the entry point for applications to use the DMT API.

The getSession methods are used to open a session on a specified subtree of the DMT. A typical way of usage:

```
serviceRef = context.getServiceReference(DmtAdmin.class.getName());
DmtAdmin admin = (DmtAdmin) context.getService(serviceRef);
DmtSession session = admin.getSession("./OSGi/Configuration");
session.createInteriorNode("./OSGi/Configuration/my.table");
```

The methods for opening a session take a node URI (the session root) as a parameter. All segments of the given URI must be within the segment length limit of the implementation, and the special characters '/' and '\' must be escaped (preceded by a '\').

See the Uri.encode(String) method for support on escaping invalid characters in a URI.

It is possible to specify a lock mode when opening the session (see lock type constants in DmtSession). This determines whether the session can run in parallel with other sessions, and the kinds of operations that can be performed in the session. All Management Objects constituting the device management tree must support read operations on their nodes, while support for write operations depends on the Management Object. Management Objects supporting write access may support transactional write, non-transactional write or both. Users of DmtAdmin should consult the Management Object specification and implementation for the supported update modes. If Management Object definition permits, implementations are encouraged to support both update modes.

117.14.3.1 public DmtSession getSession(String subtreeUri) throws DmtException

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

□ Opens a DmtSession for local usage on a given subtree of the DMT with non transactional write lock. This call is equivalent to the following: getSession(null, subtreeUri, DmtSession.LOCK_TYPE_EXCLUSIVE)

The subtreeUri parameter must contain an absolute URI. It can also be null, in this case the session is opened with the default session root, ".", that gives access to the whole tree.

To perform this operation the caller must have DmtPermission for the subtreeUri node with the Get action present.

Returns a DmtSession object for the requested subtree

Throws DmtException– with the following possible error codes:

- INVALID_URI if subtreeUri is syntactically invalid
- URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if subtreeUri specifies a non-existing node
- SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session
- COMMAND_FAILED if subtreeUri specifies a relative URI, or some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have DmtPermission for the given root node with the Get action present

117.14.3.2 public DmtSession getSession(String subtreeUri, int lockMode) throws DmtException

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

lockMode one of the lock modes specified in DmtSession

□ Opens a DmtSession for local usage on a specific DMT subtree with a given lock mode. This call is equivalent to the following: getSession(null, subtreeUri, lockMode)

The subtreeUri parameter must contain an absolute URI. It can also be null, in this case the session is opened with the default session root, ".", that gives access to the whole tree.

To perform this operation the caller must have DmtPermission for the subtreeUri node with the Get action present.

Returns a DmtSession object for the requested subtree

Throws DmtException– with the following possible error codes:

- INVALID_URI if subtreeUri is syntactically invalid
- URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if subtreeUri specifies a non-existing node
- FEATURE_NOT_SUPPORTED if atomic sessions are not supported by the implementation and lockMode requests an atomic session
- SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session

- COMMAND_FAILED if subtreeUri specifies a relative URI, if lockMode is unknown, or some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have DmtPermission for the given root node with the Get action present

117.14.3.3 public DmtSession getSession(String principal, String subtreeUri, int lockMode) throws DmtException

principal the identifier of the remote server on whose behalf the data manipulation is performed, or null for local sessions

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

lockMode one of the lock modes specified in DmtSession

□ Opens a DmtSession on a specific DMT subtree using a specific lock mode on behalf of a remote principal. If local management applications are using this method then they should provide null as the first parameter. Alternatively they can use other forms of this method without providing a principal string.

The subtreeUri parameter must contain an absolute URI. It can also be null, in this case the session is opened with the default session root, ".", that gives access to the whole tree.

This method is guarded by DmtPrincipalPermission in case of remote sessions. In addition, the caller must have Get access rights (ACL in case of remote sessions, DmtPermission in case of local sessions) on the subtreeUri node to perform this operation.

Returns a DmtSession object for the requested subtree

Throws DmtException– with the following possible error codes:

- INVALID_URI if subtreeUri is syntactically invalid
- URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if subtreeUri specifies a non-existing node
- PERMISSION_DENIED if principal is not null and the ACL of the node does not allow the Get operation for the principal on the given root node
- FEATURE_NOT_SUPPORTED if atomic sessions are not supported by the implementation and lockMode requests an atomic session
- SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session
- COMMAND_FAILED if subtreeUri specifies a relative URI, if lockMode is unknown, or some unspecified error is encountered while attempting to complete the command

SecurityException– in case of remote sessions, if the caller does not have the required DmtPrincipalPermission with a target matching the principal parameter, or in case of local sessions, if the caller does not have DmtPermission for the given root node with the Get action present

117.14.4 public class DmtConstants

Defines standard names for DmtAdmin.

Since 2.0

117.14.4.1 public static final String DDF_LIST = "org.osgi/1.0/LIST"

A string defining a DDF URI, indicating that the node is a LIST node.

117.14.4.2 public static final String DDF_MAP = "org.osgi/1.0/MAP"

A string defining a DDF URI, indicating that the node is a MAP node.

117.14.4.3 public static final String DDF_SCAFFOLD = "org.osgi/1.0/SCAFFOLD"

A string defining a DDF URI, indicating that the node is a SCAFFOLD node.

117.14.4.4 public static final String EVENT_PROPERTY_NEW_NODES = "newnodes"

A string defining the property key for the newnodes property in node related events.

117.14.4.5 public static final String EVENT_PROPERTY_NODES = "nodes"

A string defining the property key for the @{code nodes} property in node related events.

117.14.4.6 public static final String EVENT_PROPERTY_SESSION_ID = "session.id"

A string defining the property key for the session.id property in node related events.

117.14.4.7 public static final String EVENT_TOPIC_ADDED = "org/osgi/service/dmt/DmtEvent/ADDED"

A string defining the topic for the event that is sent for added nodes.

117.14.4.8 public static final String EVENT_TOPIC_COPIED = "org/osgi/service/dmt/DmtEvent/COPIED"

A string defining the topic for the event that is sent for copied nodes.

117.14.4.9 public static final String EVENT_TOPIC_DELETED = "org/osgi/service/dmt/DmtEvent/DELETED"

A string defining the topic for the event that is sent for deleted nodes.

117.14.4.10 public static final String EVENT_TOPIC_RENAMED = "org/osgi/service/dmt/DmtEvent/RENAMED"

A string defining the topic for the event that is sent for renamed nodes.

117.14.4.11 public static final String EVENT_TOPIC_REPLACED = "org/osgi/service/dmt/DmtEvent/REPLACED"

A string defining the topic for the event that is sent for replaced nodes.

117.14.4.12 public static final String EVENT_TOPIC_SESSION_CLOSED = "org/osgi/service/dmt/DmtEvent/SESSION_CLOSED"

A string defining the topic for the event that is sent for a closed session.

117.14.4.13 public static final String EVENT_TOPIC_SESSION_OPENED = "org/osgi/service/dmt/DmtEvent/SESSION_OPENED"

A string defining the topic for the event that is sent for a newly opened session.

117.14.5 public final class DmtData

An immutable data structure representing the contents of a leaf or interior node. This structure represents only the value and the format property of the node, all other properties (like MIME type) can be set and read using the DmtSession interface.

Different constructors are available to create nodes with different formats. Nodes of null format can be created using the static NULL_VALUE constant instance of this class.

FORMAT_RAW_BINARY and FORMAT_RAW_STRING enable the support of future data formats. When using these formats, the actual format name is specified as a String. The application is responsible for the proper encoding of the data according to the specified format.

Concurrency Immutable

117.14.5.1 public static final DmtData FALSE_VALUE

Constant instance representing a boolean false value.

Since 2.0

117.14.5.2 public static final int FORMAT_BASE64 = 128

The node holds an OMA DM b64 value. Like FORMAT_BINARY, this format is also represented by the Java byte[] type, the difference is only in the corresponding OMA DM format. This format does not affect the internal storage format of the data as byte[]. It is intended as a hint for the external representation of this data. Protocol Adapters can use this hint for their further processing.

117.14.5.3 public static final int FORMAT_BINARY = 64

The node holds an OMA DM bin value. The value of the node corresponds to the Java byte[] type.

117.14.5.4 public static final int FORMAT_BOOLEAN = 8

The node holds an OMA DM bool value.

117.14.5.5 public static final int FORMAT_DATE = 16

The node holds an OMA DM date value.

117.14.5.6 public static final int FORMAT_DATE_TIME = 16384

The node holds a Date object. If the getTime() equals zero then the date time is not known. If the getTime() is negative it must be interpreted as a relative number of milliseconds.

Since 2.0

117.14.5.7 public static final int FORMAT_FLOAT = 2

The node holds an OMA DM float value.

117.14.5.8 public static final int FORMAT_INTEGER = 1

The node holds an OMA DM int value.

117.14.5.9 public static final int FORMAT_LONG = 8192

The node holds a long value. The getFormatName() method can be used to get the actual format name.

Since 2.0

117.14.5.10 public static final int FORMAT_NODE = 1024

Format specifier of an internal node. An interior node can hold a Java object as value (see DmtData.DmtData(Object) and DmtData.getNode()). This value can be used by Java programs that know a specific URI understands the associated Java type. This type is further used as a return value of the MetaNode.getFormat() method for interior nodes.

117.14.5.11 public static final int FORMAT_NULL = 512

The node holds an OMA DM null value. This corresponds to the Java null type.

117.14.5.12 public static final int FORMAT_RAW_BINARY = 4096

The node holds raw protocol data encoded in binary format. The getFormatName() method can be used to get the actual format name.

117.14.5.13 public static final int FORMAT_RAW_STRING = 2048

The node holds raw protocol data encoded as String. The getFormatName() method can be used to get the actual format name.

117.14.5.14 public static final int FORMAT_STRING = 4

The node holds an OMA DM chr value.

117.14.5.15 public static final int FORMAT_TIME = 32

The node holds an OMA DM time value.

117.14.5.16 public static final int FORMAT_XML = 256

The node holds an OMA DM xml value.

117.14.5.17 public static final DmtData NULL_VALUE

Constant instance representing a leaf node of null format.

117.14.5.18 public static final DmtData TRUE_VALUE

Constant instance representing a boolean true value.

Since 2.0

117.14.5.19 public DmtData(String string)

string the string value to set

□ Create a DmtData instance of chr format with the given string value. The null string argument is valid.

117.14.5.20 public DmtData(Date date)

date the Date object to set

□ Create a DmtData instance of dateTime format with the given Date value. The given Date value must be a non-null Date object.

117.14.5.21 public DmtData(Object complex)

complex the complex data object to set

□ Create a DmtData instance of node format with the given object value. The value represents complex data associated with an interior node.

Certain interior nodes can support access to their subtrees through such complex values, making it simpler to retrieve or update all leaf nodes in a subtree.

The given value must be a non-null immutable object.

117.14.5.22 public DmtData(String value, int format)

value the string, XML, date, or time value to set

format the format of the DmtData instance to be created, must be one of the formats specified above

□ Create a DmtData instance of the specified format and set its value based on the given string. Only the following string-based formats can be created using this constructor:

- FORMAT_STRING - value can be any string
- FORMAT_XML - value must contain an XML fragment (the validity is not checked by this constructor)
- FORMAT_DATE - value must be parsable to an ISO 8601 calendar date in complete representation, basic format (pattern CCYYMMDD)
- FORMAT_TIME - value must be parsable to an ISO 8601 time of day in either local time, complete representation, basic format (pattern hhmmss) or Coordinated Universal Time, basic format (pattern hhmmssZ)

∗ The null string argument is only valid if the format is string or XML.

Throws IllegalArgumentException– if format is not one of the allowed formats, or value is not a valid string for the given format

NullPointerException– if a string, XML, date, or time is constructed and value is null

117.14.5.23 public DmtData(int integer)

integer the integer value to set

□ Create a DmtData instance of int format and set its value.

117.14.5.24 public DmtData(float flt)

flt the float value to set

□ Create a DmtData instance of float format and set its value.

117.14.5.25 public DmtData(long lng)

lng the long value to set

□ Create a DmtData instance of long format and set its value.

Since 2.0

117.14.5.26 public DmtData(boolean bool)

bool the boolean value to set

□ Create a DmtData instance of bool format and set its value.

117.14.5.27 public DmtData(byte[] bytes)

bytes the byte array to set, must not be null

□ Create a DmtData instance of bin format and set its value.

Throws NullPointerException– if bytes is null

117.14.5.28 public DmtData(byte[] bytes, boolean base64)

bytes the byte array to set, must not be null

base64 if true, the new instance will have b64 format, if false, it will have bin format

□ Create a DmtData instance of bin or b64 format and set its value. The chosen format is specified by the base64 parameter.

Throws NullPointerException– if bytes is null

117.14.5.29 public DmtData(byte[] bytes, int format)

bytes the byte array to set, must not be null

format the format of the DmtData instance to be created, must be one of the formats specified above

□ Create a DmtData instance of the specified format and set its value based on the given byte[]. Only the following byte[] based formats can be created using this constructor:

- FORMAT_BINARY
- FORMAT_BASE64

Throws IllegalArgumentException– if format is not one of the allowed formats

NullPointerException– if bytes is null

117.14.5.30 public DmtData(String formatName, String data)

formatName the name of the format, must not be null

data the data encoded according to the specified format, must not be null

□ Create a DmtData instance in FORMAT_RAW_STRING format. The data is provided encoded as a String. The actual data format is specified in formatName. The encoding used in data must conform to this format.

Throws NullPointerException– if formatName or data is null

117.14.5.31 public DmtData(String formatName, byte[] data)

formatName the name of the format, must not be null

data the data encoded according to the specified format, must not be null

□ Create a DmtData instance in FORMAT_RAW_BINARY format. The data is provided encoded as binary. The actual data format is specified in formatName. The encoding used in data must conform to this format.

Throws NullPointerException– if formatName or data is null

117.14.5.32 public boolean equals(Object obj)

obj the object to compare with this DmtData

□ Compares the specified object with this DmtData instance. Two DmtData objects are considered equal if their format is the same, and their data (selected by the format) is equal.

In case of FORMAT_RAW_BINARY and FORMAT_RAW_STRING the textual name of the data format - as returned by getFormatName() - must be equal as well.

Returns true if the argument represents the same DmtData as this object

117.14.5.33 public byte[] getBase64()

□ Gets the value of a node with base 64 (b64) format.

Returns the binary value

Throws DmtIllegalStateException– if the format of the node is not base 64.

117.14.5.34 public byte[] getBinary()

□ Gets the value of a node with binary (bin) format.

Returns the binary value

Throws DmtIllegalStateException– if the format of the node is not binary

117.14.5.35 public boolean getBoolean()

□ Gets the value of a node with boolean (bool) format.

Returns the boolean value

Throws DmtIllegalStateException– if the format of the node is not boolean

117.14.5.36 public String getDate()

□ Gets the value of a node with date format. The returned date string is formatted according to the ISO 8601 definition of a calendar date in complete representation, basic format (pattern CCYYMMDD).

Returns the date value

Throws DmtIllegalStateException– if the format of the node is not date

117.14.5.37 public Date getDateTime()

□ Gets the value of a node with dateTime format.

Returns the Date value

Throws DmtIllegalStateException– if the format of the node is not time

Since 2.0

117.14.5.38 public float getFloat()

□ Gets the value of a node with float format.

Returns the float value

Throws DmtIllegalStateException– if the format of the node is not float

117.14.5.39 public int getFormat()

□ Get the node's format, expressed in terms of type constants defined in this class. Note that the 'format' term is a legacy from OMA DM, it is more customary to think of this as 'type'.

Returns the format of the node

117.14.5.40 public String getFormatName()

□ Returns the format of this DmtData as String. For the predefined data formats this is the OMA DM defined name of the format. For FORMAT_RAW_STRING and FORMAT_RAW_BINARY this is the format specified when the object was created.

Returns the format name as String

117.14.5.41 public int getInt()

□ Gets the value of a node with integer (int) format.

Returns the integer value

Throws DmtIllegalStateException– if the format of the node is not integer

117.14.5.42 public long getLong()

□ Gets the value of a node with long format.

Returns the long value

Throws DmtIllegalStateException– if the format of the node is not long

Since 2.0

117.14.5.43 public Object getNode()

□ Gets the complex data associated with an interior node (node format).

Certain interior nodes can support access to their subtrees through complex values, making it simpler to retrieve or update all leaf nodes in the subtree.

Returns the data object associated with an interior node

Throws DmtIllegalStateException– if the format of the data is not node

117.14.5.44 public byte[] getRawBinary()

□ Gets the value of a node in raw binary (FORMAT_RAW_BINARY) format.

Returns the data value in raw binary format

Throws DmtIllegalStateException– if the format of the node is not raw binary

117.14.5.45 public String getRawString()

□ Gets the value of a node in raw String (FORMAT_RAW_STRING) format.

Returns the data value in raw String format

Throws DmtIllegalStateException– if the format of the node is not raw String

117.14.5.46 public int getSize()

□ Get the size of the data. The returned value depends on the format of data in the node:

- FORMAT_STRING, FORMAT_XML, FORMAT_BINARY, FORMAT_BASE64, FORMAT_RAW_STRING, and FORMAT_RAW_BINARY: the length of the stored data, or 0 if the data is null
- FORMAT_INTEGER and FORMAT_FLOAT: 4
- FORMAT_LONG and FORMAT_DATE_TIME: 8
- FORMAT_DATE and FORMAT_TIME: the length of the date or time in its string representation
- FORMAT_BOOLEAN: 1
- FORMAT_NODE: -1 (unknown)
- FORMAT_NULL: 0

Returns the size of the data stored by this object

117.14.5.47 public String getString()

□ Gets the value of a node with string (chr) format.

Returns the string value

Throws DmtIllegalStateException– if the format of the node is not string

117.14.5.48 public String getTime()

□ Gets the value of a node with time format. The returned time string is formatted according to the ISO 8601 definition of the time of day. The exact format depends on the value the object was initialized with: either local time, complete representation, basic format (pattern hhmmss) or Coordinated Universal Time, basic format (pattern hhmmssZ).

Returns the time value

Throws DmtIllegalStateException– if the format of the node is not time

117.14.5.49 public String getXml()

□ Gets the value of a node with xml format.

Returns the XML value

Throws DmtIllegalStateException– if the format of the node is not xml

117.14.5.50 public int hashCode()

□ Returns the hash code value for this DmtData instance. The hash code is calculated based on the data (selected by the format) of this object.

Returns the hash code value for this object

117.14.5.51 public String toString()

□ Gets the string representation of the DmtData. This method works for all formats.

For string format data - including FORMAT_RAW_STRING - the string value itself is returned, while for XML, date, time, integer, float, boolean, long and node formats the string form of the value is returned. Binary - including FORMAT_RAW_BINARY - base64 data is represented by two-digit hexadecimal numbers for each byte separated by spaces. The NULL_VALUE data has the string form of " null". Data of string or XML format containing the Java null value is represented by an empty string. DateTime data is formatted as yyyy-MM-dd'T'HH:mm:SS'Z').

Returns the string representation of this DmtData instance

117.14.6 public interface DmtEvent

Event class storing the details of a change in the tree. DmtEvent is used by DmtAdmin to notify registered EventListeners services about important changes. Events are generated after every successful DMT change, and also when sessions are opened or closed. If a DmtSession is opened in atomic mode, DMT events are only sent when the session is committed, when the changes are actually performed.

The type of the event describes the change that triggered the event delivery. Each event carries the unique identifier of the session in which the described change happened or -1 when the change originated outside a session. The events describing changes in the DMT carry the list of affected nodes. In case of COPIED or RENAMED events, the event carries the list of new nodes as well.

117.14.6.1 public static final int ADDED = 1

Event type indicating nodes that were added.

117.14.6.2 public static final int COPIED = 2

Event type indicating nodes that were copied.

117.14.6.3 public static final int DELETED = 4

Event type indicating nodes that were deleted.

117.14.6.4 public static final int RENAMED = 8

Event type indicating nodes that were renamed.

117.14.6.5 public static final int REPLACED = 16

Event type indicating nodes that were replaced.

117.14.6.6 public static final int SESSION_CLOSED = 64

Event type indicating that a session was closed. This type of event is sent when the session is closed by the client or becomes inactive for any other reason (session timeout, fatal errors in business methods, etc.).

117.14.6.7 public static final int SESSION_OPENED = 32

Event type indicating that a new session was opened.

117.14.6.8 public String[] getNewNodes()

□ This method can be used to query the new nodes, when the type of the event is COPIED or RENAMED. For all other event types this method returns null.

The array returned by this method runs parallel to the array returned by getNodes(), the elements in the two arrays contain the source and destination URIs for the renamed or copied nodes in the same order. All returned URIs are absolute.

This method returns only those nodes where the caller has the GET permission for the source or destination node of the operation. Therefore, it is possible that the method returns an empty array.

Returns the array of newly created nodes

117.14.6.9 public String[] getNodes()

□ This method can be used to query the subject nodes of this event. The method returns null for SESSION_OPENED and SESSION_CLOSED.

The method returns only those affected nodes that the caller has the GET permission for (or in case of COPIED or RENAMED events, where the caller has GET permissions for either the source or the destination nodes). Therefore, it is possible that the method returns an empty array. All returned URIs are absolute.

Returns the array of affected nodes

See Also getNewNodes()

117.14.6.10 public Object getProperty(String key)

key the name of the requested property

□ This method can be used to get the value of a single event property.

Returns the requested property value or null, if the key is not contained in the properties

See Also getPropertyNames()

Since 2.0

117.14.6.11 public String[] getPropertyNames()

□ This method can be used to query the names of all properties of this event.

The returned names can be used as key value in subsequent calls to getProperty(String).

Returns the array of property names

See Also getProperty(String)

Since 2.0

117.14.6.12 public int getSessionId()

□ This method returns the identifier of the session in which this event took place. The ID is guaranteed to be unique on a machine.

For events that do not result from a session, the session id is -1.

The availability of a session.id can also be check by using `getProperty()` with "session.id" as key.

Returns the unique identifier of the session that triggered the event or -1 if there is no session associated

117.14.6.13 public int getType()

□ This method returns the type of this event.

Returns the type of this event.

117.14.7 public interface DmtEventListener

Registered implementations of this class are notified via DmtEvent objects about important changes in the tree. Events are generated after every successful DMT change, and also when sessions are opened or closed. If a DmtSession is opened in atomic mode, DMT events are only sent when the session is committed, when the changes are actually performed.

Dmt Event Listener services must have permission DmtPermission.GET for the nodes in the `nodes` and `newNodes` property in the Dmt Event.

117.14.7.1 public static final String FILTER_EVENT = "osgi.filter.event"

A number of event types packed in a bitmap. If this service property is provided with a Dmt Event Listener service registration than that listener must only receive events where one of the Dmt Event types occur in the bitmap. The type of this service property must be `Integer`.

117.14.7.2 public static final String FILTER_PRINCIPAL = "osgi.filter.principal"

A number of names of principals. If this service property is provided with a Dmt Event Listener service registration than that listener must only receive events for which at least one of the given principals has `Get` rights. The type of this service property is `String+`.

117.14.7.3 public static final String FILTER_SUBTREE = "osgi.filter.subtree"

A number of sub-tree top nodes that define the scope of the Dmt Event Listener. If this service property is registered then the service must only receive events for nodes that are part of one of the subtrees. The type of this service property is String+.

117.14.7.4 public void changeOccurred(DmtEvent event)

event the DmtEvent describing the change in detail

□ DmtAdmin uses this method to notify the registered listeners about the change. This method is called asynchronously from the actual event occurrence.

117.14.8 public class DmtException extends Exception

Checked exception received when a DMT operation fails. Beside the exception message, a DmtException always contains an error code (one of the constants specified in this class), and may optionally contain the URI of the related node, and information about the cause of the exception.

Some of the error codes defined in this class have a corresponding error code defined in OMA DM, in these cases the name and numerical value from OMA DM is used. Error codes without counterparts in OMA DM were given numbers from a different range, starting from 1.

The cause of the exception (if specified) can either be a single Throwable instance, or a list of such instances if several problems occurred during the execution of a method. An example for the latter is the close method of DmtSession that tries to close multiple plugins, and has to report the exceptions of all failures.

Each constructor has two variants, one accepts a String node URI, the other accepts a String[] node path. The former is used by the DmtAdmin implementation, the latter by the plugins, who receive the node URI as an array of segment names. The constructors are otherwise identical.

Getter methods are provided to retrieve the values of the additional parameters, and the printStackTrace(PrintWriter) method is extended to print the stack trace of all causing throwables as well.

117.14.8.1 public static final int ALERT_NOT_ROUTED = 5

An alert can not be sent from the device to the given principal. This can happen if there is no Remote Alert Sender willing to forward the alert to the given principal, or if no principal was given and the DmtAdmin did not find an appropriate default destination.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.14.8.2 public static final int COMMAND_FAILED = 500

The recipient encountered an error which prevented it from fulfilling the request.

This error code is only used in situations not covered by any of the other error codes that a method may use. Some methods specify more specific error situations for this code, but it can generally be used for any unexpected condition that causes the command to fail.

This error code corresponds to the OMA DM response status code 500 "Command Failed".

117.14.8.3 public static final int COMMAND_NOT_ALLOWED = 405

The requested command is not allowed on the target node. This includes the following situations:

- an interior node operation is requested for a leaf node, or vice versa (e.g. trying to retrieve the children of a leaf node)
- an attempt is made to create a node where the parent is a leaf node

- an attempt is made to rename or delete the root node of the tree
- an attempt is made to rename or delete the root node of the session
- a write operation (other than setting the ACL) is performed in a non-atomic write session on a node provided by a plugin that is read-only or does not support non-atomic writing
- a node is copied to its descendant
- the ACL of the root node is changed not to include Add rights for all principals

This error code corresponds to the OMA DM response status code 405 "Command not allowed".

117.14.8.4 public static final int CONCURRENT_ACCESS = 4

An error occurred related to concurrent access of nodes. This can happen for example if a configuration node was deleted directly through the Configuration Admin service, while the node was manipulated via the tree.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.14.8.5 public static final int DATA_STORE_FAILURE = 510

An error related to the recipient data store occurred while processing the request. This error code may be thrown by any of the methods accessing the tree, but whether it is really used depends on the implementation, and the data store it uses.

This error code corresponds to the OMA DM response status code 510 "Data store failure".

117.14.8.6 public static final int FEATURE_NOT_SUPPORTED = 406

The requested command failed because an optional feature required by the command is not supported. For example, opening an atomic session might return this error code if the DmtAdmin implementation does not support transactions. Similarly, accessing the optional node properties (Title, Timestamp, Version, Size) might not succeed if either the DmtAdmin implementation or the underlying plugin does not support the property.

When getting or setting values for interior nodes (an optional optimization feature), a plugin can use this error code to indicate that the given interior node does not support values.

This error code corresponds to the OMA DM response status code 406 "Optional feature not supported".

117.14.8.7 public static final int INVALID_URI = 3

The requested command failed because the target URI or node name is `null` or syntactically invalid. This covers the following cases:

- the URI or node name ends with the '\' or '/' character
- the URI is an empty string (only invalid if the method does not accept relative URIs)
- the URI contains the segment "." at a position other than the beginning of the URI
- the node name is ".." or the URI contains such a segment
- the node name contains an unescaped '/' character

See the Uri.encode(String) method for support on escaping invalid characters in a URI.

This code is only used if the URI or node name does not match any of the criteria for URI_TOO_LONG. This error code does not correspond to any OMA DM response status code. It should be translated to the code 404 "Not Found" when transferring over OMA DM.

117.14.8.8 public static final int LIMIT_EXCEEDED = 413

The requested operation failed because a specific limit was exceeded, e.g. if a requested resource exceeds a size limit.

This error code corresponds to the OMA DM response status code 413 "Request entity too large".

Since 2.0

117.14.8.9 public static final int METADATA_MISMATCH = 2

Operation failed because of meta data restrictions. This covers any attempted deviation from the parameters defined by the MetaNode objects of the affected nodes, for example in the following situations:

- creating, deleting or renaming a permanent node, or modifying its type
- creating an interior node where the meta-node defines it as a leaf, or vice versa
- any operation on a node which does not have the required access type (e.g. executing a node that lacks the MetaNode.CMD_EXECUTE access type)
- any node creation or deletion that would violate the cardinality constraints
- any leaf node value setting that would violate the allowed formats, values, mime types, etc.
- any node creation that would violate the allowed node names

This error code can also be used to indicate any other meta data violation, even if it cannot be described by the MetaNode class. For example, detecting a multi-node constraint violation while committing an atomic session should result in this error.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 405 "Command not allowed" when transferring over OMA DM.

117.14.8.10 public static final int NODE_ALREADY_EXISTS = 418

The requested node creation operation failed because the target already exists. This can occur if the node is created directly (with one of the create... methods), or indirectly (during a copy operation).

This error code corresponds to the OMA DM response status code 418 "Already exists".

117.14.8.11 public static final int NODE_NOT_FOUND = 404

The requested target node was not found. No indication is given as to whether this is a temporary or permanent condition, unless otherwise noted.

This is only used when the requested node name is valid, otherwise the more specific error codes URI_TOO_LONG or INVALID_URI are used. This error code corresponds to the OMA DM response status code 404 "Not Found".

117.14.8.12 public static final int PERMISSION_DENIED = 425

The requested command failed because the principal associated with the session does not have adequate access control permissions (ACL) on the target. This can only appear in case of remote sessions, i.e. if the session is associated with an authenticated principal.

This error code corresponds to the OMA DM response status code 425 "Permission denied".

117.14.8.13 public static final int REMOTE_ERROR = 1

A device initiated remote operation failed. This is used when the protocol adapter fails to send an alert for any reason.

Alert routing errors (that occur while looking for the proper protocol adapter to use) are indicated by ALERT_NOT_ROUTED, this code is only for errors encountered while sending the routed alert. This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.14.8.14 public static final int ROLLBACK_FAILED = 516

The rollback command was not completed successfully. The tree might be in an inconsistent state after this error.

This error code corresponds to the OMA DM response status code 516 "Atomic roll back failed".

117.14.8.15 public static final int SESSION_CREATION_TIMEOUT = 7

Creation of a session timed out because of another ongoing session. The length of time while the DmtAdmin waits for the blocking session(s) to finish is implementation dependent.

This error code does not correspond to any OMA DM response status code. OMA has several status codes related to timeout, but these are meant to be used when a request times out, not if a session can not be established. This error code should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.14.8.16 public static final int TRANSACTION_ERROR = 6

A transaction-related error occurred in an atomic session. This error is caused by one of the following situations:

- an updating method within an atomic session can not be executed because the underlying plug-in is read-only or does not support atomic writing
- a commit operation at the end of an atomic session failed because one of the underlying plugins failed to close

The latter case may leave the tree in an inconsistent state due to the lack of a two-phase commit system, see DmtSession.commit() for details.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.14.8.17 public static final int UNAUTHORIZED = 401

The originator's authentication credentials specify a principal with insufficient rights to complete the command.

This status code is used as response to device originated sessions if the remote management server cannot authorize the device to perform the requested operation.

This error code corresponds to the OMA DM response status code 401 "Unauthorized".

117.14.8.18 public static final int URI_TOO_LONG = 414

The requested command failed because the target URI is too long for what the recipient is able or willing to process.

This error code corresponds to the OMA DM response status code 414 "URI too long".

See Also OSGi Service Platform, Mobile Specification Release 4

117.14.8.19 public DmtException(String uri, int code, String message)

uri the node on which the failed DMT operation was issued, or `null` if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or `null` if there is no error message

□ Create an instance of the exception. The `uri` and `message` parameters are optional. No originating exception is specified.

117.14.8.20 public DmtException(String uri, int code, String message, Throwable cause)

uri the node on which the failed DMT operation was issued, or `null` if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or null if there is no error message

cause the originating exception, or null if there is no originating exception

□ Create an instance of the exception, specifying the cause exception. The uri, message and cause parameters are optional.

117.14.8.21 public DmtException(String uri, int code, String message, Vector<? extends Throwable> causes, boolean fatal)

uri the node on which the failed DMT operation was issued, or null if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or null if there is no error message

causes the list of originating exceptions, or empty list or null if there are no originating exceptions

fatal whether the exception is fatal

□ Create an instance of the exception, specifying the list of cause exceptions and whether the exception is a fatal one. This constructor is meant to be used by plugins wishing to indicate that a serious error occurred which should invalidate the ongoing atomic session. The uri, message and causes parameters are optional.

If a fatal exception is thrown, no further business methods will be called on the originator plugin. In case of atomic sessions, all other open plugins will be rolled back automatically, except if the fatal exception was thrown during commit.

117.14.8.22 public DmtException(String[] path, int code, String message)

path the path of the node on which the failed DMT operation was issued, or null if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or null if there is no error message

□ Create an instance of the exception, specifying the target node as an array of path segments. This method behaves in exactly the same way as if the path was given as a URI string.

See Also DmtException(String, int, String)

117.14.8.23 public DmtException(String[] path, int code, String message, Throwable cause)

path the path of the node on which the failed DMT operation was issued, or null if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or null if there is no error message

cause the originating exception, or null if there is no originating exception

□ Create an instance of the exception, specifying the target node as an array of path segments, and specifying the cause exception. This method behaves in exactly the same way as if the path was given as a URI string.

See Also DmtException(String, int, String, Throwable)

117.14.8.24 public DmtException(String[] path, int code, String message, Vector<? extends Throwable> causes, boolean fatal)

path the path of the node on which the failed DMT operation was issued, or null if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or null if there is no error message

causes the list of originating exceptions, or empty list or null if there are no originating exceptions

fatal whether the exception is fatal

□ Create an instance of the exception, specifying the target node as an array of path segments, the list of cause exceptions, and whether the exception is a fatal one. This method behaves in exactly the same way as if the path was given as a URI string.

See Also DmtException(String, int, String, Vector, boolean)

117.14.8.25 public Throwable getCause()

□ Get the cause of this exception. Returns non-null, if this exception is caused by one or more other exceptions (like a NullPointerException in a DmtPlugin). If there are more than one cause exceptions, the first one is returned.

Returns the cause of this exception, or null if no cause was given

117.14.8.26 public Throwable[] getCauses()

□ Get all causes of this exception. Returns the causing exceptions in an array. If no cause was specified, an empty array is returned.

Returns the list of causes of this exception

117.14.8.27 public int getCode()

□ Get the error code associated with this exception. Most of the error codes within this exception correspond to OMA DM error codes.

Returns the error code

117.14.8.28 public String getMessage()

□ Get the message associated with this exception. The returned string also contains the associated URI (if any) and the exception code. The resulting message has the following format (parts in square brackets are only included if the field inside them is not null):

```
<exception_code>[: '<uri>'][: <error_message>]
```

Returns the error message in the format described above

117.14.8.29 public String getURI()

□ Get the node on which the failed DMT operation was issued. Some operations like DmtSession.close() don't require an URI, in this case this method returns null.

Returns the URI of the node, or null

117.14.8.30 public boolean isFatal()

□ Check whether this exception is marked as fatal in the session. Fatal exceptions trigger an automatic rollback of atomic sessions.

Returns whether the exception is marked as fatal

117.14.8.31 public void printStackTrace(PrintStream s)

s PrintStream to use for output

□ Prints the exception and its stacktrace to the specified print stream. Any causes that were specified for this exception are also printed, together with their stacktraces.

117.14.9 public class DmtIllegalStateException extends RuntimeException

Unchecked illegal state exception. This class is used in DMT because java.lang.IllegalStateException does not exist in CLDC.

117.14.9.1 public DmtIllegalStateException()

□ Create an instance of the exception with no message.

117.14.9.2 public DmtIllegalStateException(String message)

message the reason for the exception

□ Create an instance of the exception with the specified message.

117.14.9.3 public DmtIllegalStateException(Throwable cause)

cause the cause of the exception

□ Create an instance of the exception with the specified cause exception and no message.

117.14.9.4 public DmtIllegalStateException(String message, Throwable cause)

message the reason for the exception

cause the cause of the exception

□ Create an instance of the exception with the specified message and cause exception.

117.14.10 public interface DmtSession

DmtSession provides concurrent access to the DMT. All DMT manipulation commands for management applications are available on the DmtSession interface. The session is associated with a root node which limits the subtree in which the operations can be executed within this session.

Most of the operations take a node URI as parameter, which can be either an absolute URI (starting with "./") or a URI relative to the root node of the session. The empty string as relative URI means the root URI the session was opened with. All segments of a URI must be within the segment length limit of the implementation, and the special characters '/' and '\' must be escaped (preceded by a '\').

See the Uri.encode(String) method for support on escaping invalid characters in a URI.

If the URI specified does not correspond to a legitimate node in the tree an exception is thrown. The only exception is the isNodeUri(String) method which returns false in case of an invalid URI.

Each method of DmtSession that accesses the tree in any way can throw DmtIllegalStateException if the session has been closed or invalidated (due to timeout, fatal exceptions, or unexpectedly unregistered plugins).

117.14.10.1 public static final int LOCK_TYPE_ATOMIC = 2

LOCK_TYPE_ATOMIC is an exclusive lock with transactional functionality. Commands of an atomic session will either fail or succeed together, if a single command fails then the whole session will be rolled back.

117.14.10.2 public static final int LOCK_TYPE_EXCLUSIVE = 1

LOCK_TYPE_EXCLUSIVE lock guarantees full access to the tree, but can not be shared with any other locks.

117.14.10.3 public static final int LOCK_TYPE_SHARED = 0

Sessions created with LOCK_TYPE_SHARED lock allows read-only access to the tree, but can be shared between multiple readers.

117.14.10.4 public static final int STATE_CLOSED = 1

The session is closed, DMT manipulation operations are not available, they throw DmtIllegalState-Exception if tried.

117.14.10.5 public static final int STATE_INVALID = 2

The session is invalid because a fatal error happened. Fatal errors include the timeout of the session, any DmtException with the 'fatal' flag set, or the case when a plugin service is unregistered while in use by the session. DMT manipulation operations are not available, they throw DmtIllegalStateException if tried.

117.14.10.6 public static final int STATE_OPEN = 0

The session is open, all session operations are available.

117.14.10.7 public void close() throws DmtException

□ Closes a session. If the session was opened with atomic lock mode, the DmtSession must first persist the changes made to the DMT by calling commit() on all (transactional) plugins participating in the session. See the documentation of the commit() method for details and possible errors during this operation.

The state of the session changes to DmtSession.STATE_CLOSED if the close operation completed successfully, otherwise it becomes DmtSession.STATE_INVALID.

Throws DmtException– with the following possible error codes:

- METADATA_MISMATCH in case of atomic sessions, if the commit operation failed because of meta-data restrictions
- CONCURRENT_ACCESS in case of atomic sessions, if the commit operation failed because of some modification outside the scope of the DMT to the nodes affected in the session
- TRANSACTION_ERROR in case of atomic sessions, if an underlying plugin failed to commit
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if an underlying plugin failed to close, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.14.10.8 public void commit() throws DmtException

□ Commits a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent commit() and rollback() calls.

This method can fail even if all operations were successful. This can happen due to some multi-node semantic constraints defined by a specific implementation. For example, node A can be required to always have children A/B, A/C and A/D. If this condition is broken when commit() is executed, the method will fail, and throw a METADATA_MISMATCH exception.

An error situation can arise due to the lack of a two phase commit mechanism in the underlying plugins. As an example, if plugin A has committed successfully but plugin B failed, the whole session must fail, but there is no way to undo the commit performed by A. To provide predictable behavior, the commit operation should continue with the remaining plugins even after detecting a failure. All exceptions received from failed commits are aggregated into one TRANSACTION_ERROR exception thrown by this method.

In many cases the tree is not the only way to manage a given part of the system. It may happen that while modifying some nodes in an atomic session, the underlying settings are modified in parallel outside the scope of the DMT. If this is detected during commit, an exception with the code CONCURRENT_ACCESS is thrown.

Throws DmtException– with the following possible error codes:

- METADATA_MISMATCH if the operation failed because of meta-data restrictions
- CONCURRENT_ACCESS if it is detected that some modification has been made outside the scope of the DMT to the nodes affected in the session's operations
- TRANSACTION_ERROR if an error occurred during the commit of any of the underlying plugins
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was not opened using the LOCK_TYPE_ATOMIC lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.14.10.9 public void copy(String nodeUri, String newNodeUri, boolean recursive) throws DmtException

nodeUri the node or root of a subtree to be copied

newNodeUri the URI of the new node or root of a subtree

recursive false if only a single node is copied, true if the whole subtree is copied

□ Create a copy of a node or a whole subtree. Beside the structure and values of the nodes, most properties are also copied, with the exception of the ACL (Access Control List), Timestamp and Version properties.

The copy method is essentially a convenience method that could be substituted with a sequence of retrieval and update operations. This determines the permissions required for copying. However, some optimization can be possible if the source and target nodes are all handled by DmtAdmin or by the same plugin. In this case, the handler might be able to perform the underlying management operation more efficiently: for example, a configuration table can be copied at once instead of reading each node for each entry and creating it in the new tree.

This method may result in any of the errors possible for the contributing operations. Most of these are collected in the exception descriptions below, but for the full list also consult the documentation of getChildNodeNames(String), isLeafNode(String), getNodeValue(String), getNodeType(String), getNodeTitle(String), setNodeTitle(String, String), createLeafNode(String, DmtData, String) and createInteriorNode(String, String).

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri or newNodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node, or if newNodeUri points to a node that cannot exist in the tree according to the meta-data (see getMetaNode(String))
- NODE_ALREADY_EXISTS if newNodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the copied node(s) does not allow the Get operation, or the ACL of the parent of the target node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if nodeUri is an ancestor of newNodeUri, or if any of the implied retrieval or update operations are not allowed
- METADATA_MISMATCH if any of the meta-data constraints of the implied retrieval or update operations are violated

- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if either URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the copied node(s) with the Get action present, or for the parent of the target node with the Add action

117.14.10.10 public void createInteriorNode(String nodeUri) throws DmtException

nodeUri the URI of the node to create

□ Create an interior node. If the parent node does not exist, it is created automatically, as if this method were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for the node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent interior node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

117.14.10.11 public void createInteriorNode(String nodeUri, String type) throws DmtException

nodeUri the URI of the node to create

type the type URI of the interior node, can be null if no node type is defined

□ Create an interior node with a given type. The type of interior node, if specified, is a URI identifying a DDF document. If the parent node does not exist, it is created automatically, as if createInteriorNode(String) were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for the node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent interior node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Interior node type identifiers must follow the format defined in section 7.7.7.2 of the OMA Device Management Tree and Description document. Checking the validity of the type string does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, if the type string is invalid (see above), or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

See Also createInteriorNode(String), OMA Device Management Tree and Description v1.2 draft [http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-TS-DM-TND-V1_2-20050615-C.zip]

117.14.10.12 public void createLeafNode(String nodeUri) throws DmtException

nodeUri the URI of the node to create

□ Create a leaf node with default value and MIME type. If a node does not have a default value or MIME type, this method will throw a DmtException with error code METADATA_MISMATCH. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if createInteriorNode(String) were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

See Also createLeafNode(String, DmtData)

117.14.10.13 public void createLeafNode(String nodeUri, DmtData value) throws DmtException

nodeUri the URI of the node to create

value the value to be given to the new node, can be null

□ Create a leaf node with a given value and the default MIME type. If the specified value is null, the default value is taken. If the node does not have a default MIME type or value (if needed), this method will throw a DmtException with error code METADATA_MISMATCH. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if createInteriorNode(String) were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, the node value must conform to the value constraints, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Nodes of null format can be created by using DmtData.NULL_VALUE as second argument.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

117.14.10.14 public void createLeafNode(String nodeUri, DmtData value, String mimeType) throws DmtException

nodeUri the URI of the node to create

value the value to be given to the new node, can be null

mimeType the MIME type to be given to the new node, can be null

□ Create a leaf node with a given value and MIME type. If the specified value or MIME type is null, their default values are taken. If the node does not have the necessary defaults, this method will throw a DmtException with error code METADATA_MISMATCH. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if createInteriorNode(String) were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, the node value must conform to the value constraints, the MIME type must be among the listed types, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Nodes of null format can be created by using DmtData.NULL_VALUE as second argument.

The MIME type string must conform to the definition in RFC 2045. Checking its validity does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
- COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, if mimeType is not a proper MIME type string (see above), or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

See Also createLeafNode(String, DmtData), RFC 2045 [http://www.ietf.org/rfc/rfc2045.txt]

117.14.10.15 public void deleteNode(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Delete the given node. Deleting interior nodes is recursive, the whole subtree under the given node is deleted. It is not allowed to delete the root node of the session.

If meta-data is available for a node, several checks are made before deleting it. The node must be non-permanent, it must have the MetaNode.CMD_DELETE access type, and if zero occurrences of the node are not allowed, it must not be the last one.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Delete operation for the associated principal
- COMMAND_NOT_ALLOWED if the target node is the root of the session, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be deleted because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Delete action present

117.14.10.16 public void execute(String nodeUri, String data) throws DmtException

nodeUri the node on which the execute operation is issued

data the parameter of the execute operation, can be null

□ Executes a node. This corresponds to the EXEC operation in OMA DM. This method cannot be called in a read-only session.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if the node does not exist
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Execute operation for the associated principal
- COMMAND_NOT_ALLOWED if the specified node is a scaffold node
- METADATA_MISMATCH if the node cannot be executed according to the meta-data (does not have MetaNode.CMD_EXECUTE access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, if no DmtExecPlugin is associated with the node and the DmtAdmin can not execute the node, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Exec action present

See Also execute(String, String, String)

117.14.10.17 public void execute(String nodeUri, String correlator, String data) throws DmtException

nodeUri the node on which the execute operation is issued

correlator an identifier to associate this operation with any notifications sent in response to it, can be null if not needed

data the parameter of the execute operation, can be null

□ Executes a node, also specifying a correlation ID for use in response notifications. This operation corresponds to the EXEC command in OMA DM. This method cannot be called in a read-only session.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued. If a correlation ID is specified, it should be used as the correlator parameter for notifications sent in response to this execute operation.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if the node does not exist
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Execute operation for the associated principal
- COMMAND_NOT_ALLOWED if the specified node is a scaffold node
- METADATA_MISMATCH if the node cannot be executed according to the meta-data (does not have MetaNode.CMD_EXECUTE access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, if no DmtExecPlugin is associated with the node, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Exec action present

See Also execute(String, String)

117.14.10.18 public String[] getChildNodeNames(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the list of children names of a node. The returned array contains the names - not the URIs - of the immediate children nodes of the given node. The elements are in no particular order. The returned array must not contain null entries.

Returns the list of child node names as a string array or an empty string array if the node has no children

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid

- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- COMMAND_NOT_ALLOWED if the specified node is not an interior node
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.19 public Acl getEffectiveNodeAcl(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Gives the Access Control List in effect for a given node. The returned Acl takes inheritance into account, that is if there is no ACL defined for the node, it will be derived from the closest ancestor having an ACL defined.

Returns the Access Control List belonging to the node

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (the node does not have MetaNode.CMD_GET access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

See Also getNodeAcl(String)

117.14.10.20 public int getLockType()

□ Gives the type of lock the session has.

Returns the lock type of the session, one of LOCK_TYPE_SHARED, LOCK_TYPE_EXCLUSIVE and LOCK_TYPE_ATOMIC

117.14.10.21 public MetaNode getMetaNode(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the meta data which describes a given node. Meta data can only be inspected, it can not be changed.

The MetaNode object returned to the client is the combination of the meta data returned by the data plugin (if any) plus the meta data returned by the DmtAdmin. If there are differences in the meta data elements known by the plugin and the DmtAdmin then the plugin specific elements take precedence.

Note, that a node does not have to exist for having meta-data associated with it. This method may provide meta-data for any node that can possibly exist in the tree (any node defined in the specification). For nodes that are not defined, it may throw DmtException with the error code NODE_NOT_FOUND. To allow easier implementation of plugins that do not provide meta-data, it is allowed to return null for any node, regardless of whether it is defined or not.

Returns a MetaNode which describes meta data information, can be null if there is no meta data available for the given node

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a node that is not defined in the tree (see above)
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.22 public Acl getNodeAcl(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the Access Control List associated with a given node. The returned Acl object does not take inheritance into account, it gives the ACL specifically given to the node.

Returns the Access Control List belonging to the node or null if none defined

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (the node does not have MetaNode.CMD_GET access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

See Also getEffectiveNodeAcl(String)

117.14.10.23 public int getNodeSize(String nodeUri) throws DmtException

nodeUri the URI of the leaf node

□ Get the size of the data in a leaf node. The returned value depends on the format of the data in the node, see the description of the DmtData.getSize() method for the definition of node size for each format.

Returns the size of the data in the node

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- COMMAND_NOT_ALLOWED if the specified node is not a leaf node
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- FEATURE_NOT_SUPPORTED if the Size property is not supported by the DmtAdmin implementation or the underlying plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

See Also DmtData.getSize()

117.14.10.24 public Date getNodeTimestamp(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the timestamp when the node was created or last modified.

Returns the timestamp of the last modification

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- FEATURE_NOT_SUPPORTED if the Timestamp property is not supported by the DmtAdmin implementation or the underlying plugin

- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.25 public String getNodeTitle(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the title of a node. There might be no title property set for a node.

Returns the title of the node, or null if the node has no title

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- FEATURE_NOT_SUPPORTED if the Title property is not supported by the DmtAdmin implementation or the underlying plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.26 public String getNodeType(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document; a null type means that there is no DDF document overriding the tree structure defined by the ancestors.

Returns the type of the node, can be null

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)

- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.27 public DmtData getNodeValue(String nodeUri) throws DmtException

nodeUri the URI of the node to retrieve

□ Get the data contained in a leaf or interior node. When retrieving the value associated with an interior node, the caller must have rights to read all nodes in the subtree under the given node.

Returns the data of the node, can not be null

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node (and the ACLs of all its descendants in case of interior nodes) do not allow the Get operation for the associated principal
- METADATA_MISMATCH if the node value cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node (and all its descendants in case of interior nodes) with the Get action present

117.14.10.28 public int getNodeVersion(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the version of a node. The version can not be set, it is calculated automatically by the device. It is incremented modulo 0x10000 at every modification of the value or any other property of the node, for both leaf and interior nodes. When a node is created the initial value is 0.

Returns the version of the node

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal

- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- FEATURE_NOT_SUPPORTED if the Version property is not supported by the DmtAdmin implementation or the underlying plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.29 public String getPrincipal()

□ Gives the name of the principal on whose behalf the session was created. Local sessions do not have an associated principal, in this case null is returned.

Returns the identifier of the remote server that initiated the session, or null for local sessions

117.14.10.30 public String getRootUri()

□ Get the root URI associated with this session. Gives "." if the session was created without specifying a root, which means that the target of this session is the whole DMT.

Returns the root URI

117.14.10.31 public int getSessionId()

□ The unique identifier of the session. The ID is generated automatically, and it is guaranteed to be unique on a machine for a specific Dmt Admin. A session id must be larger than 0.

Returns the session identification number

117.14.10.32 public int getState()

□ Get the current state of this session.

Returns the state of the session, one of STATE_OPEN, STATE_CLOSED and STATE_INVALID

117.14.10.33 public boolean isLeafNode(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Tells whether a node is a leaf or an interior node of the DMT.

Returns true if the given node is a leaf node

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
- METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.34 public boolean isNodeUri(String nodeUri)

nodeUri the URI to check

□ Check whether the specified URI corresponds to a valid node in the DMT.

Returns true if the given node exists in the DMT

Throws DmtIllegalStateException– if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.14.10.35 public void renameNode(String nodeUri, String newName) throws DmtException

nodeUri the URI of the node to rename

newName the new name property of the node

□ Rename a node. This operation only changes the name of the node (updating the timestamp and version properties if they are supported), the value and the other properties are not changed. The new name of the node must be provided, the new URI is constructed from the base of the old URI and the given name. It is not allowed to rename the root node of the session.

If available, the meta-data of the original and the new nodes are checked before performing the rename operation. Neither node can be permanent, their leaf/interior property must match, and the name change must not violate any of the cardinality constraints. The original node must have the MetaNode.CMD_REPLACE access type, and the name of the new node must conform to the valid names.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri or newName is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node, or if the new node is not defined in the tree according to the meta-data (see getMetaNode(String))
- NODE_ALREADY_EXISTS if there already exists a sibling of nodeUri with the name newName
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED if the target node is the root of the session, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node could not be renamed because of meta-data restrictions (see above)
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

117.14.10.36 public void rollback() throws DmtException

□ Rolls back a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent commit() and rollback() calls.

Throws DmtException– with the error code ROLLBACK_FAILED in case the rollback did not succeed

DmtIllegalStateException– if the session was not opened using the LOCK_TYPE_ATOMIC lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.14.10.37 public void setDefaultNodeValue(String nodeUri) throws DmtException

nodeUri the URI of the node

□ Set the value of a leaf or interior node to its default. The default can be defined by the node's MetaNode. The method throws a METADATA_MISMATCH exception if the node does not have a default value.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node is permanent or cannot be modified according to the metadata (does not have the MetaNode.CMD_REPLACE access type), or if there is no default value defined for this node
- FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

See Also setNodeValue(String, DmtData)

117.14.10.38 public void setNodeAcl(String nodeUri, Acl acl) throws DmtException

nodeUri the URI of the node

acl the Access Control List to be set on the node, can be null

□ Set the Access Control List associated with a given node. To perform this operation, the caller needs to have replace rights (Acl.REPLACE or the corresponding Java permission depending on the session type) as described below:

- if nodeUri specifies a leaf node, replace rights are needed on the parent of the node
- if nodeUri specifies an interior node, replace rights on either the node or its parent are sufficient

If the given acl is null or an empty ACL (not specifying any permissions for any principals), then the ACL of the node is deleted, and the node will inherit the ACL from its parent node.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node or its parent (see above) does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED if the command attempts to set the ACL of the root node not to include Add rights for all principals
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– in case of local sessions, if the caller does not have DmtPermission for the node or its parent (see above) with the Replace action present

117.14.10.39 public void setNodeTitle(String nodeUri, String title) throws DmtException

nodeUri the URI of the node

title the title text of the node, can be null

□ Set the title property of a node. The length of the title string in UTF-8 encoding must not exceed 255 bytes.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type)
- FEATURE_NOT_SUPPORTED if the Title property is not supported by the DmtAdmin implementation or the underlying plugin
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store

- COMMAND_FAILED if the title string is too long, if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

117.14.10.40 public void setNodeType(String nodeUri, String type) throws DmtException

nodeUri the URI of the node

type the type of the node, can be null

□ Set the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document.

For interior nodes, a null type string means that there is no DDF document overriding the tree structure defined by the ancestors. For leaf nodes, it requests that the default MIME type is used for the given node. If the node does not have a default MIME type this method will throw a DmtException with error code METADATA_MISMATCH. Note that a node might have a default MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

MIME types must conform to the definition in RFC 2045. Interior node type identifiers must follow the format defined in section 7.7.7.2 of the OMA Device Management Tree and Description document. Checking the validity of the type string does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node is permanent or cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type), and in case of leaf nodes, if null is given and there is no default MIME type, or the given MIME type is not allowed
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, if the type string is invalid (see above), or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

See Also RFC 2045 [http://www.ietf.org/rfc/rfc2045.txt], OMA Device Management Tree and Description v1.2 draft [http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-TS-DM-TND-V1_2-20050615-C.zip]

117.14.10.41 public void setNodeValue(String nodeUri, DmtData data) throws DmtException

nodeUri the URI of the node

data the data to be set, can be null

□ Set the value of a leaf or interior node. The format of the node is contained in the DmtData object. For interior nodes, the format must be FORMAT_NODE, while for leaf nodes this format must not be used.

If the specified value is null, the default value is taken. In this case, if the node does not have a default value, this method will throw a DmtException with error code METADATA_MISMATCH. Nodes of null format can be set by using DmtData.NULL_VALUE as second argument.

An Event of type REPLACE is sent out for a leaf node. A replaced interior node sends out events for each of its children in depth first order and node names sorted with Arrays.sort(String[]). When setting a value on an interior node, the values of the leaf nodes under it can change, but the structure of the subtree is not modified by the operation.

Throws DmtException– with the following possible error codes:

- INVALID_URI if nodeUri is null or syntactically invalid
- URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
- NODE_NOT_FOUND if nodeUri points to a non-existing node
- PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
- COMMAND_NOT_ALLOWED if the given data has FORMAT_NODE format but the node is a leaf node (or vice versa), or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
- METADATA_MISMATCH if the node is permanent or cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type), or if the given value does not conform to the meta-data value constraints
- FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
- TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException– if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

117.14.11 public interface MetaNode

The MetaNode contains meta data as standardized by OMA DM but extends it (without breaking the compatibility) to provide for better DMT data quality in an environment where many software components manipulate this data.

The interface has several types of functions to describe the nodes in the DMT. Some methods can be used to retrieve standard OMA DM metadata such as access type, cardinality, default, etc., others are for data extensions such as valid names and values. In some cases the standard behavior has been extended, for example it is possible to provide several valid MIME types, or to differentiate between normal and automatic dynamic nodes.

Most methods in this interface receive no input, just return information about some aspect of the node. However, there are two methods that behave differently, isValidName(String) and isValidValue(DmtData). These validation methods are given a potential node name or value (respectively), and can decide whether it is valid for the given node. Passing the validation methods is a necessary condition for a name or value to be used, but it is not necessarily sufficient: the plugin may carry out more thorough (more expensive) checks when the node is actually created or set.

If a MetaNode is available for a node, the DmtAdmin must use the information provided by it to filter out invalid requests on that node. However, not all methods on this interface are actually used for this purpose, as many of them (e.g. getFormat() or getValidNames()) can be substituted with the validating methods. For example, isValidValue(DmtData) can be expected to check the format, minimum, maximum, etc. of a given value, making it unnecessary for the DmtAdmin to call getFormat(), getMin(), getMax() etc. separately. It is indicated in the description of each method if the DmtAdmin does not enforce the constraints defined by it - such methods are only for external use, for example in user interfaces.

Most of the methods of this class return null if a certain piece of meta information is not defined for the node or providing this information is not supported. Methods of this class do not throw exceptions.

117.14.11.1 public static final int AUTOMATIC = 2

Constant for representing an automatic node in the tree. This must be returned by getScope(). AUTOMATIC nodes are part of the life cycle of their parent node, they usually describe attributes/properties of the parent.

117.14.11.2 public static final int CMD_ADD = 0

Constant for the ADD access type. If can(int) returns true for this operation, this node can potentially be added to its parent. Nodes with PERMANENT or AUTOMATIC scope typically do not have this access type.

117.14.11.3 public static final int CMD_DELETE = 1

Constant for the DELETE access type. If can(int) returns true for this operation, the node can potentially be deleted.

117.14.11.4 public static final int CMD_EXECUTE = 2

Constant for the EXECUTE access type. If can(int) returns true for this operation, the node can potentially be executed.

117.14.11.5 public static final int CMD_GET = 4

Constant for the GET access type. If can(int) returns true for this operation, the value, the list of child nodes (in case of interior nodes) and the properties of the node can potentially be retrieved.

117.14.11.6 public static final int CMD_REPLACE = 3

Constant for the REPLACE access type. If can(int) returns true for this operation, the value and other properties of the node can potentially be modified.

117.14.11.7 public static final int DYNAMIC = 1

Constant for representing a dynamic node in the tree. This must be returned by getScope(). Dynamic nodes can be added and deleted.

117.14.11.8 public static final int PERMANENT = 0

Constant for representing a PERMANENT node in the tree. This must be returned by getScope() if the node cannot be added, deleted or modified in any way through tree operations. PERMANENT nodes in general map to the roots of Plugins.

117.14.11.9 public boolean can(int operation)

operation One of the MetaNode.CMD_... constants.

□ Check whether the given operation is valid for this node. If no meta-data is provided for a node, all operations are valid.

Returns false if the operation is not valid for this node or the operation code is not one of the allowed constants

117.14.11.10 public DmtData getDefault()

□ Get the default value of this node if any.

Returns The default value or null if not defined

117.14.11.11 public String getDescription()

□ Get the explanation string associated with this node. Can be null if no description is provided for this node.

Returns node description string or null for no description

117.14.11.12 public Object getExtensionProperty(String key)

key the key for the extension property

□ Returns the value for the specified extension property key. This method only works if the provider of this MetaNode provides proprietary extensions to node meta data.

Returns the value of the requested property, cannot be null

Throws IllegalArgumentException– if the specified key is not supported by this MetaNode

117.14.11.13 public String[] getExtensionPropertyKeys()

□ Returns the list of extension property keys, if the provider of this MetaNode provides proprietary extensions to node meta data. The method returns null if the node doesn't provide such extensions.

Returns the array of supported extension property keys

117.14.11.14 public int getFormat()

□ Get the node's format, expressed in terms of type constants defined in DmtData. If there are multiple formats allowed for the node then the format constants are OR-ed. Interior nodes must have DmtData.FORMAT_NODE format, and this code must not be returned for leaf nodes. If no meta-data is provided for a node, all applicable formats are considered valid (with the above constraints regarding interior and leaf nodes).

Note that the 'format' term is a legacy from OMA DM, it is more customary to think of this as 'type'.

The formats returned by this method are not checked by DmtAdmin, they are only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed format(s) of the node

117.14.11.15 public double getMax()

□ Get the maximum allowed value associated with a node of numeric format. If no meta-data is provided for a node, there is no upper limit to its value. This method is only meaningful if the node has one of the numeric formats: integer, float, or long format. The returned limit has double type, as this can be used to denote all numeric limits with full precision. The actual maximum should be the largest integer, float or long number that does not exceed the returned value.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed maximum, or `Double.MAX_VALUE` if there is no upper limit defined or the node's format is not one of the numeric formats integer, float, or long

117.14.11.16 public int getMaxOccurrence()

□ Get the number of maximum occurrences of this type of nodes on the same level in the DMT. Returns `Integer.MAX_VALUE` if there is no upper limit. Note that if the occurrence is greater than 1 then this node can not have siblings with different metadata. In other words, if different types of nodes coexist on the same level, their occurrence can not be greater than 1. If no meta-data is provided for a node, there is no upper limit on the number of occurrences.

Returns The maximum allowed occurrence of this node type

117.14.11.17 public String[] getMimeTypes()

□ Get the list of MIME types this node can hold. The first element of the returned list must be the default MIME type.

All MIME types are considered valid if no meta-data is provided for a node or if `null` is returned by this method. In this case the default MIME type cannot be retrieved from the meta-data, but the node may still have a default. This hidden default (if it exists) can be utilized by passing `null` as the type parameter of DmtSession.setNodeType(String, String) or DmtSession.createLeafNode(String, DmtData, String).

Returns the list of allowed MIME types for this node, starting with the default MIME type, or `null` if all types are allowed

117.14.11.18 public double getMin()

□ Get the minimum allowed value associated with a node of numeric format. If no meta-data is provided for a node, there is no lower limit to its value. This method is only meaningful if the node has one of the numeric formats: integer, float, or long format. The returned limit has `double` type, as this can be used to denote both integer and float limits with full precision. The actual minimum should be the smallest integer, float or long value that is equal or larger than the returned value.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed minimum, or `Double.MIN_VALUE` if there is no lower limit defined or the node's format is not one of the numeric formats integer, float, or long

117.14.11.19 public String[] getRawFormatNames()

□ Get the format names for any raw formats supported by the node. This method is only meaningful if the list of supported formats returned by getFormat() contains DmtData.FORMAT_RAW_STRING or DmtData.FORMAT_RAW_BINARY: it specifies precisely which raw format(s) are actually supported. If the node cannot contain data in one of the raw types, this method must return `null`.

The format names returned by this method are not checked by DmtAdmin, they are only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed format name(s) of raw data stored by the node, or `null` if raw formats are not supported

117.14.11.20 public int getScope()

□ Return the scope of the node. Valid values are MetaNode.PERMANENT, MetaNode.DYNAMIC and MetaNode.AUTOMATIC. Note that a permanent node is not the same as a node where the DELETE operation is not allowed. Permanent nodes never can be deleted, whereas a non-deletable node can disappear in a recursive DELETE operation issued on one of its parents. If no meta-data is provided for a node, it can be assumed to be a dynamic node.

Returns PERMANENT for permanent nodes, AUTOMATIC for nodes that are automatically created, and DYNAMIC otherwise

117.14.11.21 public String[] getValidNames()

□ Return an array of Strings if valid names are defined for the node, or null if no valid name list is defined or if this piece of meta info is not supported. If no meta-data is provided for a node, all names are considered valid.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidName(String) for checking the name, its behavior should be consistent with this method.

Returns the valid values for this node name, or null if not defined

117.14.11.22 public DmtData[] getValidValues()

□ Return an array of DmtData objects if valid values are defined for the node, or null otherwise. If no meta-data is provided for a node, all values are considered valid.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the valid values for this node, or null if not defined

117.14.11.23 public boolean isLeaf()

□ Check whether the node is a leaf node or an internal one.

Returns true if the node is a leaf node

117.14.11.24 public boolean isValidName(String name)

name the node name to check for validity

□ Checks whether the given name is a valid name for this node. This method can be used for example to ensure that the node name is always one of a predefined set of valid names, or that it matches a specific pattern. This method should be consistent with the values returned by getValidNames() (if any), the DmtAdmin only calls this method for name validation.

This method may return true even if not all aspects of the name have been checked, expensive operations (for example those that require external resources) need not be performed here. The actual node creation may still indicate that the node name is invalid.

Returns false if the specified name is found to be invalid for the node described by this meta-node, true otherwise

117.14.11.25 public boolean isValidValue(DmtData value)

value the value to check for validity

□ Checks whether the given value is valid for this node. This method can be used to ensure that the value has the correct format and range, that it is well formed, etc. This method should be consistent with the constraints defined by the getFormat(), getValidValues(), getMin() and getMax() methods (if applicable), as the Dmt Admin only calls this method for value validation.

This method may return true even if not all aspects of the value have been checked, expensive operations (for example those that require external resources) need not be performed here. The actual value setting method may still indicate that the value is invalid.

Returns false if the specified value is found to be invalid for the node described by this meta-node, true otherwise

117.14.11.26 public boolean isZeroOccurrenceAllowed()

□ Check whether zero occurrence of this node is valid. If no meta-data is returned for a node, zero occurrences are allowed.

Returns true if zero occurrence of this node is valid

117.14.12 public final class Uri

This class contains static utility methods to manipulate DMT URIs.

Syntax of valid DMT URIs:

- A slash ('/' \u002F) is the separator of the node names. Slashes used in node name must therefore be escaped using a backslash slash ("\/"). The backslash must be escaped with a double backslash sequence. A backslash found must be ignored when it is not followed by a slash or backslash.
- The node name can be constructed using full Unicode character set (except the Supplementary code, not being supported by CLDC/CDC). However, using the full Unicode character set for node names is discouraged because the encoding in the underlying storage as well as the encoding needed in communications can create significant performance and memory usage overhead. Names that are restricted to the URI set [-a-zA-Z0-9_.!~*'()] are most efficient.
- URIs used in the DMT must be treated and interpreted as case sensitive.
- No End Slash: URI must not end with the delimiter slash ('/' \u002F). This implies that the root node must be denoted as "." and not "./".
- No parent denotation: URI must not be constructed using the character sequence "../" to traverse the tree upwards.
- Single Root: The character sequence "./" must not be used anywhere else but in the beginning of a URI.

117.14.12.1 public static final String PATH_SEPARATOR = "/"

This constant stands for a string identifying the path separator in the DmTree ("/").

Since 2.0

117.14.12.2 public static final char PATH_SEPARATOR_CHAR = 47

This constant stands for a char identifying the path separator in the DmTree ('/').

Since 2.0

117.14.12.3 public static final String ROOT_NODE = "."

This constant stands for a string identifying the root of the DmTree (".").

Since 2.0

117.14.12.4 public static final char ROOT_NODE_CHAR = 46

This constant stands for a char identifying the root of the DmTree ('.').

Since 2.0

117.14.12.5 public static String decode(String nodeName)

nodeName the node name to be decoded

□ Decode the node name so that back slash and forward slash are unescaped from a back slash.

Returns the decoded node name

Since 2.0

117.14.12.6 public static String encode(String nodeName)

nodeName the node name to be encoded

□ Encode the node name so that back slash and forward slash are escaped with a back slash. This method is the reverse of decode(String).

Returns the encoded node name

Since 2.0

117.14.12.7 public static boolean isAbsoluteUri(String uri)

uri the URI to be checked, must not be `null` and must contain a valid URI

□ Checks whether the specified URI is an absolute URI. An absolute URI contains the complete path to a node in the DMT starting from the DMT root (".").

Returns whether the specified URI is absolute

Throws `NullPointerException`– if the specified URI is `null`

`IllegalArgumentException`– if the specified URI is malformed

117.14.12.8 public static boolean isValidUri(String uri)

uri the URI to be validated

□ Checks whether the specified URI is valid. A URI is considered valid if it meets the following constraints:

- the URI is not `null`;
- the URI follows the syntax defined for valid DMT URIs;

The exact definition of the length of a URI and its segments is given in the descriptions of the `getMaxUriLength()` and `getMaxSegmentNameLength()` methods.

Returns whether the specified URI is valid

117.14.12.9 public static String mangle(String nodeName)

nodeName the node name to be mangled (if necessary), must not be `null` or empty

□ Returns a node name that is valid for the tree operation methods, based on the given node name. This transformation is not idempotent, so it must not be called with a parameter that is the result of a previous `mangle` method call.

Node name mangling is needed in the following cases:

- if the name contains '/' or '\' characters

A node name that does not suffer from either of these problems is guaranteed to remain unchanged by this method. Therefore the client may skip the mangling if the node name is known to be valid (though it is always safe to call this method).

The method returns the normalized `nodeName` as described below. Invalid node names are normalized in different ways, depending on the cause. If the name contains '/' or '\' characters, then these are simply escaped by inserting an additional '\' before each occurrence. If the length of the name does exceed the limit, the following mechanism is used to normalize it:

- the SHA-1 digest of the name is calculated
- the digest is encoded with the base 64 algorithm
- all '/' characters in the encoded digest are replaced with '_'
- trailing '=' signs are removed

Returns the normalized node name that is valid for tree operations

Throws NullPointerException– if nodeName is null

IllegalArgumentException– if nodeName is empty

117.14.12.10 public static String[] toPath(String uri)

uri the URI to be split, must not be null

□ Split the specified URI along the path separator '/' characters and return an array of URI segments. Special characters in the returned segments are escaped. The returned array may be empty if the specified URI was empty.

Returns an array of URI segments created by splitting the specified URI

Throws NullPointerException– if the specified URI is null

IllegalArgumentException– if the specified URI is malformed

117.14.12.11 public static String toUri(String[] path)

path a possibly empty array of URI segments, must not be null

□ Construct a URI from the specified URI segments. The segments must already be mangled.

If the specified path is an empty array then an empty URI ("") is returned.

Returns the URI created from the specified segments

Throws NullPointerException– if the specified path or any of its segments are null

IllegalArgumentException– if the specified path contains too many or malformed segments or the resulting URI is too long

117.15 org.osgi.service.dmt.spi

Device Management Tree SPI Package Version 2.0.

This package contains the interface classes that compose the Device Management SPI (Service Provider Interface). These interfaces are implemented by DMT plugins; users of the DmtAdmin interface do not interact directly with these.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt.spi; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt.spi; version="[2.0,2.1)"
```

117.15.1 Summary

- DataPlugin - An implementation of this interface takes the responsibility of handling data requests in a subtree of the DMT.
- ExecPlugin - An implementation of this interface takes the responsibility of handling node execute requests in a subtree of the DMT.
- MountPlugin - This interface can be optionally implemented by a DataPlugin or ExecPlugin in order to get information about its absolute mount points in the overall DMT.

- MountPoint - This interface can be implemented to represent a single mount point.
- ReadableDataSession - Provides read-only access to the part of the tree handled by the plugin that created this session.
- ReadWriteDataSession - Provides non-atomic read-write access to the part of the tree handled by the plugin that created this session.
- TransactionalDataSession - Provides atomic read-write access to the part of the tree handled by the plugin that created this session.

117.15.2 public interface DataPlugin

An implementation of this interface takes the responsibility of handling data requests in a subtree of the DMT.

In an OSGi environment such implementations should be registered at the OSGi service registry specifying the list of root node URIs in a String array or in case of a single value as String in the dataRootURIs registration parameter.

When the first reference in a session is made to a node handled by this plugin, the DmtAdmin calls one of the open... methods to retrieve a plugin session object for processing the request. The called method depends on the lock type of the current session. In case of openReadWriteSession(String[], DmtSession) and openAtomicSession(String[], DmtSession), the plugin may return null to indicate that the specified lock type is not supported. In this case the DmtAdmin may call openReadOnlySession(String[], DmtSession) to start a read-only plugin session, which can be used as long as there are no write operations on the nodes handled by this plugin.

The sessionRoot parameter of each method is a String array containing the segments of the URI pointing to the root of the session. This is an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

117.15.2.1 public static final String DATA_ROOT_URIS = "dataRootURIs"

The string to be used as key for the "dataRootURIs" property when an DataPlugin is registered.

Since 2.0

117.15.2.2 public static final String MOUNT_POINTS = "mountPoints"

The string to be used as key for the mount points property when a DataPlugin is registered with mount points.

117.15.2.3 public TransactionalDataSession openAtomicSession(String[] sessionRoot, DmtSession session) throws DmtException

sessionRoot the path to the subtree which is locked in the current session, must not be null

session the session from which this plugin instance is accessed, must not be null

□ This method is called to signal the start of an atomic read-write session when the first reference is made within a DmtSession to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no other sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read-write operations in an atomic block, or null if the plugin does not support atomic read-write sessions

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if sessionRoot points to a non-existing node
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if some underlying operation failed because of lack of permissions

117.15.2.4 public ReadableDataSession openReadOnlySession(String[] sessionRoot, DmtSession session) throws DmtException

sessionRoot the path to the subtree which is accessed in the current session, must not be null

session the session from which this plugin instance is accessed, must not be null

□ This method is called to signal the start of a read-only session when the first reference is made within a DmtSession to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no writing sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read operations

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if sessionRoot points to a non-existing node
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if some underlying operation failed because of lack of permissions

117.15.2.5 public ReadWriteDataSession openReadWriteSession(String[] sessionRoot, DmtSession session) throws DmtException

sessionRoot the path to the subtree which is locked in the current session, must not be null

session the session from which this plugin instance is accessed, must not be null

□ This method is called to signal the start of a non-atomic read-write session when the first reference is made within a DmtSession to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no other sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read-write operations, or null if the plugin does not support non-atomic read-write sessions

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if sessionRoot points to a non-existing node
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if some underlying operation failed because of lack of permissions

117.15.3 public interface ExecPlugin

An implementation of this interface takes the responsibility of handling node execute requests in a subtree of the DMT.

In an OSGi environment such implementations should be registered at the OSGi service registry specifying the list of root node URIs in a String array or in case of a single value as String in the execRootURIs registration parameter.

117.15.3.1 public static final String EXEC_ROOT_URIS = "execRootURIs"

The string to be used as key for the "execRootURIs" property when an ExecPlugin is registered.

Since 2.0

117.15.3.2 public static final String MOUNT_POINTS = "mountPoints"

The string to be used as key for the mount points property when an Exec Plugin is registered with mount points.

117.15.3.3 public void execute(DmtSession session, String[] nodePath, String correlator, String data) throws DmtException

session a reference to the session in which the operation was issued, must not be null

nodePath the absolute path of the node to be executed, must not be null

correlator an identifier to associate this operation with any alerts sent in response to it, can be null

data the parameter of the execute operation, can be null

□ Execute the given node with the given data. This operation corresponds to the EXEC command in OMA DM.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued. Session information is given as it is needed for sending alerts back from the plugin. If a correlation ID is specified, it should be used as the correlator parameter for alerts sent in response to this execute operation.

The nodePath parameter contains an array of path segments identifying the node to be executed in the subtree of this plugin. This is an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if the node does not exist
- METADATA_MISMATCH if the command failed because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

See Also DmtSession.execute(String, String), DmtSession.execute(String, String, String)

117.15.4 public interface MountPlugin

This interface can be optionally implemented by a DataPlugin or ExecPlugin in order to get information about its absolute mount points in the overall DMT.

This is especially interesting, if the plugin is mapped to the tree as part of a list. In such a case the id for this particular data plugin is determined by the DmtAdmin after the registration of the plugin and therefore unknown to the plugin in advance.

This is not a service interface, the Data or Exec Plugin does not also have to register this interface as a service, the Dmt Admin should use an instanceof to detect that a Plugin is also a Mount Plugin.

Since 2.0

117.15.4.1 public void mountPointAdded(MountPoint mountPoint)

mountPoint the newly mapped mount point

□ Provides the MountPoint describing the path where the plugin is mapped in the overall DMT. The given mountPoint is withdrawn with the mountPointRemoved(MountPoint) method. Corresponding mount points must compare equal and have an appropriate hash code.

117.15.4.2 public void mountPointRemoved(MountPoint mountPoint)

mountPoint The unmapped mount point array of MountPoint objects that have been removed from the mapping

□ Informs the plugin that the provided MountPoint objects have been removed from the mapping. The given mountPoint is withdrawn method. Mount points must compare equal and have an appropriate hash code with the given Mount Point in mountPointAdded(MountPoint).

NOTE: attempts to invoke the postEvent method on the provided MountPoint must be ignored.

117.15.5 public interface MountPoint

This interface can be implemented to represent a single mount point.

It provides function to get the absolute mounted uri and a shortcut method to post events via the DmtAdmin.

Since 2.0

117.15.5.1 public boolean equals(Object other)

□ This object must provide a suitable hash function such that a Mount Point given in MountPlugin.mountPointAdded(MountPoint) is equal to the corresponding Mount Point in MountPlugin.mountPointRemoved(MountPoint). Object.equals(Object)

117.15.5.2 public String[] getMountPath()

□ Provides the absolute mount path of this MountPoint

Returns the absolute mount path of this MountPoint

117.15.5.3 public int hashCode()

□ This object must provide a suitable hash function such that a Mount Point given in MountPlugin.mountPointAdded(MountPoint) has the same hashCode as the corresponding Mount Point in MountPlugin.mountPointRemoved(MountPoint). Object.hashCode()

117.15.5.4 public void postEvent(String topic, String[] relativeURIs, Dictionary<String, ?> properties)

topic the topic of the event to send. Valid values are:

- org/osgi/service/dmt/DmtEvent/ADDED if the change was caused by an add action
- org/osgi/service/dmt/DmtEvent/DELETED if the change was caused by a delete action
- org/osgi/service/dmt/DmtEvent/REPLACED if the change was caused by a replace action

Must not be null.

relativeURIs an array of affected node URI's. All URI's specified here are relative to the current MountPoint's mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

properties an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain a key EVENT_PROPERTY_NODES, then the value of this property is ignored and will be overwritten by relativeURIs.

□ Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners.

Throws IllegalArgumentException– if the topic has not one of the defined values

117.15.5.5 public void postEvent(String topic, String[] relativeURIs, String[] newRelativeURIs, Dictionary<String, ?> properties)

topic the topic of the event to send. Valid values are:

- org/osgi/service/dmt/DmtEvent/RENAMED if the change was caused by a rename action
- org/osgi/service/dmt/DmtEvent/COPIED if the change was caused by a copy action

Must not be null.

relativeURIs an array of affected node URI's.

All URI's specified here are relative to the current MountPoint's mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

newRelativeURIs an array of affected node URI's. The value of this parameter determines the value of the event property EVENT_PROPERTY_NEW_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NEW_NODES property will be set to an empty array.

properties an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain the keys EVENT_PROPERTY_NODES or EVENT_PROPERTY_NEW_NODES, then the values of these properties are ignored and will be overwritten by relativeURIs and newRelativeURIs.

□ Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners.

Throws IllegalArgumentException– if the topic has not one of the defined values

117.15.6 public interface ReadableDataSession

Provides read-only access to the part of the tree handled by the plugin that created this session.

Since the ReadWriteDataSession and TransactionalDataSession interfaces inherit from this interface, some of the method descriptions do not apply for an instance that is only a ReadableDataSession. For example, the close() method description also contains information about its behavior when invoked as part of a transactional session.

The nodePath parameters appearing in this interface always contain an array of path segments identifying a node in the subtree of this plugin. This parameter contains an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Error handling

When a tree access command is called on the DmtAdmin service, it must perform an extensive set of checks on the parameters and the authority of the caller before delegating the call to a plugin. Therefore plugins can take certain circumstances for granted: that the path is valid and is within the subtree of the plugin and the session, the command can be applied to the given node (e.g. the target of getChildNodeNames is an interior node), etc. All errors described by the error codes DmtException.INVALID_URI, DmtException.URI_TOO_LONG, DmtException.PERMISSION_DENIED, DmtException.COMMAND_NOT_ALLOWED and DmtException.TRANSACTION_ERROR are fully filtered out before control reaches the plugin.

If the plugin provides meta-data for a node, the DmtAdmin service must also check the constraints specified by it, as described in MetaNode. If the plugin does not provide meta-data, it must perform the necessary checks for itself and use the DmtException.METADATA_MISMATCH error code to indicate such discrepancies.

The DmtAdmin does not check that the targeted node exists before calling the plugin. It is the responsibility of the plugin to perform this check and to throw a DmtException.NODE_NOT_FOUND if needed. In this case the DmtAdmin must pass through this exception to the caller of the corresponding DmtSession method.

The plugin can use the remaining error codes as needed. If an error does not fit into any other category, the DmtException.COMMAND_FAILED code should be used.

117.15.6.1 public void close() throws DmtException

□ Closes a session. This method is always called when the session ends for any reason: if the session is closed, if a fatal error occurs in any method, or if any error occurs during commit or rollback. In case the session was invalidated due to an exception during commit or rollback, it is guaranteed that no methods are called on the plugin until it is closed. In case the session was invalidated due to a fatal exception in one of the tree manipulation methods, only the rollback method is called before this (and only in atomic sessions).

This method should not perform any data manipulation, only cleanup operations. In non-atomic read-write sessions the data manipulation should be done instantly during each tree operation, while in atomic sessions the DmtAdmin always calls TransactionalDataSession.commit() automatically before the session is actually closed.

Throws DmtException– with the error code COMMAND_FAILED if the plugin failed to close for any reason

117.15.6.2 public String[] getChildNodeNames(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the list of children names of a node. The returned array contains the names - not the URIs - of the immediate children nodes of the given node. The returned array may contain null entries, but these are removed by the DmtAdmin before returning it to the client.

Returns the list of child node names as a string array or an empty string array if the node has no children

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.3 public MetaNode getMetaNode(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the meta data which describes a given node. Meta data can be only inspected, it can not be changed.

Meta data support by plugins is an optional feature. It can be used, for example, when a data plugin is implemented on top of a data store or another API that has their own metadata, such as a relational database, in order to avoid metadata duplication and inconsistency. The meta data specific to the plugin returned by this method is complemented by meta data from the DmtAdmin before returning it to the client. If there are differences in the meta data elements known by the plugin and the DmtAdmin then the plugin specific elements take precedence.

Note, that a node does not have to exist for having meta-data associated with it. This method may provide meta-data for any node that can possibly exist in the tree (any node defined by the Management Object provided by the plugin). For nodes that are not defined, a DmtException may be thrown with the NODE_NOT_FOUND error code. To allow easier implementation of plugins that do not provide meta-data, it is allowed to return null for any node, regardless of whether it is defined or not.

Returns a MetaNode which describes meta data information, can be null if there is no meta data available for the given node

Throws DmtException – with the following possible error codes:

- NODE_NOT_FOUND if nodeUri points to a node that is not defined in the tree (see above)
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.4 public int getNodeSize(String[] nodePath) throws DmtException

nodePath the absolute path of the leaf node

□ Get the size of the data in a leaf node. The value to return depends on the format of the data in the node, see the description of the DmtData.getSize() method for the definition of node size for each format.

Returns the size of the data in the node

Throws DmtException – with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the Size property is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtData.getSize()

117.15.6.5 public Date getNodeTimestamp(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the timestamp when the node was last modified.

Returns the timestamp of the last modification

Throws DmtException – with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the Timestamp property is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.6 public String getNodeTitle(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the title of a node. There might be no title property set for a node.

Returns the title of the node, or null if the node has no title

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the Title property is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.7 public String getNodeType(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document; a null type means that there is no DDF document overriding the tree structure defined by the ancestors.

Returns the type of the node, can be null

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.8 public DmtData getNodeValue(String[] nodePath) throws DmtException

nodePath the absolute path of the node to retrieve

□ Get the data contained in a leaf or interior node.

Returns the data of the leaf node, must not be null

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.9 public int getNodeVersion(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the version of a node. The version can not be set, it is calculated automatically by the device. It is incremented modulo 0x10000 at every modification of the value or any other property of the node, for both leaf and interior nodes. When a node is created the initial value is 0.

Returns the version of the node

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the Version property is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.10 public boolean isLeafNode(String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Tells whether a node is a leaf or an interior node of the DMT.

Returns true if the given node is a leaf node

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.6.11 public boolean isNodeUri(String[] nodePath)

nodePath the absolute path to check

□ Check whether the specified path corresponds to a valid node in the DMT.

Returns true if the given node exists in the DMT

117.15.6.12 public void nodeChanged(String[] nodePath) throws DmtException

nodePath the absolute path of the node that has changed

□ Notifies the plugin that the given node has changed outside the scope of the plugin, therefore the Version and Timestamp properties must be updated (if supported). This method is needed because the ACL property of a node is managed by the DmtAdmin instead of the plugin. The DmtAdmin must call this method whenever the ACL property of a node changes.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- DATA_STORE_FAILURE if an error occurred while accessing the data store

- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

117.15.7 public interface ReadWriteDataSession extends ReadableDataSession

Provides non-atomic read-write access to the part of the tree handled by the plugin that created this session.

The nodePath parameters appearing in this interface always contain an array of path segments identifying a node in the subtree of this plugin. This parameter contains an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Error handling

When a tree manipulation command is called on the DmtAdmin service, it must perform an extensive set of checks on the parameters and the authority of the caller before delegating the call to a plugin. Therefore plugins can take certain circumstances for granted: that the path is valid and is within the subtree of the plugin and the session, the command can be applied to the given node (e.g. the target of setNodeValue is a leaf node), etc. All errors described by the error codes DmtException.INVALID_URI, DmtException.URI_TOO_LONG, DmtException.PERMISSION_DENIED, DmtException.COMMAND_NOT_ALLOWED and DmtException.TRANSACTION_ERROR are fully filtered out before control reaches the plugin.

If the plugin provides meta-data for a node, the DmtAdmin service must also check the constraints specified by it, as described in MetaNode. If the plugin does not provide meta-data, it must perform the necessary checks for itself and use the DmtException.METADATA_MISMATCH error code to indicate such discrepancies.

The DmtAdmin does not check that the targeted node exists (or that it does not exist, in case of a node creation) before calling the plugin. It is the responsibility of the plugin to perform this check and to throw a DmtException.NODE_NOT_FOUND or DmtException.NODE_ALREADY_EXISTS if needed. In this case the DmtAdmin must pass through this exception to the caller of the corresponding DmtSession method.

The plugin can use the remaining error codes as needed. If an error does not fit into any other category, the DmtException.COMMAND_FAILED code should be used.

117.15.7.1 public void copy(String[] nodePath, String[] newNodePath, boolean recursive) throws DmtException

nodePath an absolute path specifying the node or the root of a subtree to be copied

newNodePath the absolute path of the new node or root of a subtree

recursive false if only a single node is copied, true if the whole subtree is copied

□ Create a copy of a node or a whole subtree. Beside the structure and values of the nodes, most properties managed by the plugin must also be copied, with the exception of the Timestamp and Version properties.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node, or if newNodePath points to a node that cannot exist in the tree
- NODE_ALREADY_EXISTS if newNodePath points to a node that already exists
- METADATA_MISMATCH if the node could not be copied because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the copy operation is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.copy(String, String, boolean)

117.15.7.2 public void createInteriorNode(String[] nodePath, String type) throws DmtException

nodePath the absolute path of the node to create

type the type URI of the interior node, can be null if no node type is defined

□ Create an interior node with a given type. The type of interior node, if specified, is a URI identifying a DDF document.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a node that cannot exist in the tree
- NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.createInteriorNode(String), DmtSession.createInteriorNode(String, String)

117.15.7.3 public void createLeafNode(String[] nodePath, DmtData value, String mimeType) throws DmtException

nodePath the absolute path of the node to create

value the value to be given to the new node, can be null

mimeType the MIME type to be given to the new node, can be null

□ Create a leaf node with a given value and MIME type. If the specified value or MIME type is null, their default values must be taken.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a node that cannot exist in the tree
- NODE_ALREADY_EXISTS if nodePath points to a node that already exists
- METADATA_MISMATCH if the node could not be created because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.createLeafNode(String), DmtSession.createLeafNode(String, DmtData), DmtSession.createLeafNode(String, DmtData, String)

117.15.7.4 public void deleteNode(String[] nodePath) throws DmtException

nodePath the absolute path of the node to delete

□ Delete the given node. Deleting interior nodes is recursive, the whole subtree under the given node is deleted.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node

- METADATA_MISMATCH if the node could not be deleted because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.deleteNode(String)

117.15.7.5 public void renameNode(String[] nodePath, String newName) throws DmtException

nodePath the absolute path of the node to rename

newName the new name property of the node

□ Rename a node. This operation only changes the name of the node (updating the timestamp and version properties if they are supported), the value and the other properties are not changed. The new name of the node must be provided, the new path is constructed from the base of the old path and the given name.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node, or if the new node is not defined in the tree
- NODE_ALREADY_EXISTS if there already exists a sibling of nodePath with the name newName
- METADATA_MISMATCH if the node could not be renamed because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.renameNode(String, String)

117.15.7.6 public void setNodeTitle(String[] nodePath, String title) throws DmtException

nodePath the absolute path of the node

title the title text of the node, can be null

□ Set the title property of a node. The length of the title is guaranteed not to exceed the limit of 255 bytes in UTF-8 encoding.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the title could not be set because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the Title property is not supported by the plugin
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.setNodeTitle(String, String)

117.15.7.7 public void setNodeType(String[] nodePath, String type) throws DmtException

nodePath the absolute path of the node *type* the type of the node, can be null

□ Set the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document.

For interior nodes, the null type should remove the reference (if any) to a DDF document overriding the tree structure defined by the ancestors. For leaf nodes, it requests that the default MIME type is used for the given node.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the type could not be set because of meta-data restrictions
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.setNodeType(String, String)

117.15.7.8 public void setNodeValue(String[] nodePath, DmtData data) throws DmtException

nodePath the absolute path of the node

data the data to be set, can be null

□ Set the value of a leaf or interior node. The format of the node is contained in the DmtData object. For interior nodes, the format is FORMAT_NODE, while for leaf nodes this format is never used.

If the specified value is null, the default value must be taken; if there is no default value, a DmtException with error code METADATA_MISMATCH must be thrown.

Throws DmtException– with the following possible error codes:

- NODE_NOT_FOUND if nodePath points to a non-existing node
- METADATA_MISMATCH if the value could not be set because of meta-data restrictions
- FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
- DATA_STORE_FAILURE if an error occurred while accessing the data store
- COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

SecurityException– if the caller does not have the necessary permissions to execute the underlying management operation

See Also DmtSession.setNodeValue(String, DmtData)

117.15.8 public interface TransactionalDataSession extends ReadWriteDataSession

Provides atomic read-write access to the part of the tree handled by the plugin that created this session.

117.15.8.1 public void commit() throws DmtException

□ Commits a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent commit() and rollback() calls.

This method can fail even if all operations were successful. This can happen due to some multi-node semantic constraints defined by a specific implementation. For example, node A can be required to always have children A/B, A/C and A/D. If this condition is broken when `commit()` is executed, the method will fail, and throw a `METADATA_MISMATCH` exception.

In many cases the tree is not the only way to manage a given part of the system. It may happen that while modifying some nodes in an atomic session, the underlying settings are modified in parallel outside the scope of the DMT. If this is detected during commit, an exception with the code `CONCURRENT_ACCESS` is thrown.

Throws `DmtException`– with the following possible error codes

- `METADATA_MISMATCH` if the operation failed because of meta-data restrictions
- `CONCURRENT_ACCESS` if it is detected that some modification has been made outside the scope of the DMT to the nodes affected in the session's operations
- `DATA_STORE_FAILURE` if an error occurred while accessing the data store
- `COMMAND_FAILED` if some unspecified error is encountered while attempting to complete the command

`SecurityException`– if the caller does not have the necessary permissions to execute the underlying management operation

117.15.8.2 public void rollback() throws DmtException

□ Rolls back a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent commit and rollback calls.

Throws `DmtException`– with the error code `ROLLBACK_FAILED` in case the rollback did not succeed

`SecurityException`– if the caller does not have the necessary permissions to execute the underlying management operation

117.16 org.osgi.service.dmt.notification

Device Management Tree Notification Package Version 2.0.

This package contains the public API of the Notification service. This service enables the sending of asynchronous notifications to management servers. Permission classes are provided by the `org.osgi.service.dmt.security` package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.dmt.notification; version="[2.0,3.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.dmt.notification; version="[2.0,2.1)"`

117.16.1 Summary

- `AlertItem` - Immutable data structure carried in an alert (client initiated notification).
- `NotificationService` - NotificationService enables sending asynchronous notifications to a management server.

117.16.2 public class AlertItem

Immutable data structure carried in an alert (client initiated notification). The AlertItem describes details of various notifications that can be sent by the client, for example as alerts in the OMA DM protocol. The use cases include the client sending a session request to the server (alert 1201), the client notifying the server of completion of a software update operation (alert 1226) or sending back results in response to an asynchronous EXEC command.

The data syntax and semantics varies widely between various alerts, so does the optionality of particular parameters of an alert item. If an item, such as source or type, is not defined, the corresponding getter method returns null. For example, for alert 1201 (client-initiated session) all elements will be null.

The syntax used in AlertItem class corresponds to the OMA DM alert format. NotificationService implementations on other management protocols should map these constructs to the underlying protocol.

117.16.2.1 public AlertItem(String source, String type, String mark, DmtData data)

source the URI of the node which is the source of the alert item

type a MIME type or a URN that identifies the type of the data in the alert item

data a DmtData object that contains the format and value of the data in the alert item

mark the mark parameter of the alert item

□ Create an instance of the alert item. The constructor takes all possible data entries as parameters. Any of these parameters can be null. The semantics of the parameters may be refined by the definition of a specific alert, identified by its alert code (see NotificationService.sendNotification(String, int, String, AlertItem[])). In case of Generic Alerts for example (code 1226), the mark parameter contains a severity string.

117.16.2.2 public AlertItem(String[] source, String type, String mark, DmtData data)

source the path of the node which is the source of the alert item

type a MIME type or a URN that identifies the type of the data in the alert item

data a DmtData object that contains the format and value of the data in the alert item

mark the mark parameter of the alert item

□ Create an instance of the alert item, specifying the source node URI as an array of path segments. The constructor takes all possible data entries as parameters. Any of these parameters can be null. The semantics of the parameters may be refined by the definition of a specific alert, identified by its alert code (see NotificationService.sendNotification(String, int, String, AlertItem[])). In case of Generic Alerts for example (code 1226), the mark parameter contains a severity string.

117.16.2.3 public DmtData getData()

□ Get the data associated with the alert item. The returned DmtData object contains the format and the value of the data in the alert item. There might be no data associated with the alert item.

Returns the data associated with the alert item, or null if there is no data

117.16.2.4 public String getMark()

□ Get the mark parameter associated with the alert item. The interpretation of the mark parameter depends on the alert being sent, as identified by the alert code in NotificationService.sendNotification(String, int, String, AlertItem[]) . There might be no mark associated with the alert item.

Returns the mark associated with the alert item, or null if there is no mark

117.16.2.5 public String getSource()

- Get the node which is the source of the alert. There might be no source associated with the alert item.

Returns the URI of the node which is the source of this alert, or null if there is no source

117.16.2.6 public String getType()

- Get the type associated with the alert item. The type string is a MIME type or a URN that identifies the type of the data in the alert item (returned by getData()). There might be no type associated with the alert item.

Returns the type associated with the alert item, or null if there is no type

117.16.2.7 public String toString()

- Returns the string representation of this alert item. The returned string includes all parameters of the alert item, and has the following format:

```
AlertItem(<source>, <type>, <mark>, <data>)
```

The last parameter is the string representation of the data value. The format of the data is not explicitly included.

Returns the string representation of this alert item

117.16.3 public interface NotificationService

NotificationService enables sending asynchronous notifications to a management server. The implementation of NotificationService should register itself in the OSGi service registry as a service.

117.16.3.1 public void sendNotification(String principal, int code, String correlator, AlertItem[] items) throws DmtException

principal the principal name which is the recipient of this notification, can be null

code the alert code, can be 0 if not needed

correlator optional field that contains the correlation identifier of an associated exec command, can be null if not needed

items the data of the alert items carried in this alert, can be null or empty if not needed

- Sends a notification to a named principal. It is the responsibility of the NotificationService to route the notification to the given principal using the registered org.osgi.service.dmt.notification.spi.RemoteAlertSender services.

In remotely initiated sessions the principal name identifies the remote server that created the session, this can be obtained using the session's getPrincipal call.

The principal name may be omitted if the client does not know the principal name. Even in this case the routing might be possible if the Notification Service finds an appropriate default destination (for example if it is only connected to one protocol adapter, which is only connected to one management server).

Since sending the notification and receiving acknowledgment for it is potentially a very time-consuming operation, notifications are sent asynchronously. This method should attempt to ensure that the notification can be sent successfully, and should throw an exception if it detects any problems. If the method returns without error, the notification is accepted for sending and the implementation must make a best-effort attempt to deliver it.

In case the notification is an asynchronous response to a previous execute command, a correlation identifier can be specified to provide the association between the execute and the notification.

In order to send a notification using this method, the caller must have an AlertPermission with a target string matching the specified principal name. If the principal parameter is null (the principal name is not known), the target of the AlertPermission must be "*".

When this method is called with null correlator, null or empty AlertItem array, and a 0 code as values, it should send a protocol specific default notification to initiate a management session. For example, in case of OMA DM this is alert 1201 "Client Initiated Session". The principal parameter can be used to determine the recipient of the session initiation request.

Throws DmtException– with the following possible error codes:

- UNAUTHORIZED when the remote server rejected the request due to insufficient authorization
- ALERT_NOT_ROUTED when the alert can not be routed to the given principal
- REMOTE_ERROR in case of communication problems between the device and the destination
- COMMAND_FAILED for unspecified errors encountered while attempting to complete the command
- FEATURE_NOT_SUPPORTED if the underlying management protocol doesn't support asynchronous notifications

SecurityException– if the caller does not have the required AlertPermission with a target matching the principal parameter, as described above

117.17 org.osgi.service.dmt.notification.spi

Device Management Tree Notification SPI Package Version 2.0.

This package contains the SPI (Service Provider Interface) of the Notification service. These interfaces are implemented by Protocol Adapters capable of delivering notifications to management servers on a specific protocol. Users of the NotificationService interface do not interact directly with this package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt.notification.spi; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt.notification.spi; version="[2.0,2.1)"
```

117.17.1 Summary

- RemoteAlertSender - The RemoteAlertSender can be used to send notifications to (remote) entities identified by principal names.

117.17.2 public interface RemoteAlertSender

The RemoteAlertSender can be used to send notifications to (remote) entities identified by principal names. This service is provided by Protocol Adapters, and is used by the org.osgi.service.dmt.notification.NotificationService when sending alerts. Implementations of this interface have to be able to connect and send alerts to one or more management servers in a protocol specific way.

The properties of the service registration should specify a list of destinations (principals) where the service is capable of sending alerts. This can be done by providing a String array of principal names in the principals registration property. If this property is not registered, the service will be treated as the default sender. The default alert sender is only used when a more specific alert sender cannot be found.

The principals registration property is used when the org.osgi.service.dmt.notification.NotificationService.sendNotification(String, int, String, AlertItem[]) method is called, to find the proper RemoteAlertSender for the given destination. If the caller does not specify a principal, the alert is only sent if the Notification Sender finds a default alert sender, or if the choice is unambiguous for some other reason (for example if only one alert sender is registered).

117.17.2.1 public void sendAlert(String principal, int code, String correlator, AlertItem[] items) throws Exception

principal the name identifying the server where the alert should be sent, can be null

code the alert code, can be 0 if not needed

correlator the correlation identifier of an associated EXEC command, or null if there is no associated EXEC

items the data of the alert items carried in this alert, can be empty or null if no alert items are needed

□ Sends an alert to a server identified by its principal name. In case the alert is sent in response to a previous execute command, a correlation identifier can be specified to provide the association between the execute and the alert.

The principal parameter specifies which server the alert should be sent to. This parameter can be null if the client does not know the name of the destination. The alert should still be delivered if possible; for example if the alert sender is only connected to one destination.

Any exception thrown on this method will be propagated to the original sender of the event, wrapped in a DmtException with the code REMOTE_ERROR.

Since sending the alert and receiving acknowledgment for it is potentially a very time-consuming operation, alerts are sent asynchronously. This method should attempt to ensure that the alert can be sent successfully, and should throw an exception if it detects any problems. If the method returns without error, the alert is accepted for sending and the implementation must make a best-effort attempt to deliver it.

Throws Exception– if the alert can not be sent to the server

117.18 org.osgi.service.dmt.security

Device Management Tree Security Package Version 2.0.

This package contains the permission classes used by the Device Management API in environments that support the Java 2 security model.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt.security; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt.security; version="[2.0,2.1)"
```

117.18.1 Summary

- AlertPermission - Indicates the callers authority to send alerts to management servers, identified by their principal names.

- DmtPermission - Controls access to management objects in the Device Management Tree (DMT).
- DmtPrincipalPermission - Indicates the callers authority to create DMT sessions on behalf of a remote management server.

117.18.2 public class AlertPermission extends Permission

Indicates the callers authority to send alerts to management servers, identified by their principal names.

AlertPermission has a target string which controls the principal names where alerts can be sent. A wildcard is allowed at the end of the target string, to allow sending alerts to any principal with a name matching the given prefix. The "*" target means that alerts can be sent to any destination.

117.18.2.1 public AlertPermission(String target)

target the name of a principal, can end with * to match any principal identifier with the given prefix

□ Creates a new AlertPermission object with its name set to the target string. Name must be non-null and non-empty.

Throws NullPointerException– if name is null

IllegalArgumentException– if name is empty

117.18.2.2 public AlertPermission(String target, String actions)

target the name of the server, can end with * to match any server identifier with the given prefix

actions no actions defined, must be "*" for forward compatibility

□ Creates a new AlertPermission object using the 'canonical' two argument constructor. In this version this class does not define any actions, the second argument of this constructor must be "*" so that this class can later be extended in a backward compatible way.

Throws NullPointerException– if name or actions is null

IllegalArgumentException– if name is empty or actions is not "*"

117.18.2.3 public boolean equals(Object obj)

obj the object to compare to this AlertPermission instance

□ Checks whether the given object is equal to this AlertPermission instance. Two AlertPermission instances are equal if they have the same target string.

Returns true if the parameter represents the same permissions as this instance

117.18.2.4 public String getActions()

□ Returns the action list (always * in the current version).

Returns the action string "*"

117.18.2.5 public int hashCode()

□ Returns the hash code for this permission object. If two AlertPermission objects are equal according to the equals(Object) method, then calling this method on each of the two AlertPermission objects must produce the same integer result.

Returns hash code for this permission object

117.18.2.6 public boolean implies(Permission p)

p the permission to check for implication

□ Checks if this AlertPermission object implies the specified permission. Another AlertPermission instance is implied by this permission either if the target strings are identical, or if this target can be made identical to the other target by replacing a trailing "*" with any string.

Returns true if this AlertPermission instance implies the specified permission

117.18.2.7 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object for storing AlertPermission objects.

Returns the new PermissionCollection

117.18.3 public class DmtPermission extends Permission

Controls access to management objects in the Device Management Tree (DMT). It is intended to control local access to the DMT. DmtPermission target string identifies the management object URI and the action field lists the OMA DM commands that are permitted on the management object. Example:

```
DmtPermission("./OSGi/bundles", "Add,Replace,Get");
```

This means that owner of this permission can execute Add, Replace and Get commands on the ./OSGi/bundles management object. It is possible to use wildcards in both the target and the actions field. Wildcard in the target field means that the owner of the permission can access children nodes of the target node. Example:

```
DmtPermission("./OSGi/bundles/*", "Get");
```

This means that owner of this permission has Get access on every child node of ./OSGi/bundles. The asterisk does not necessarily have to follow a '/' character. For example the "./OSGi/a*" target matches the ./OSGi/applications subtree.

If wildcard is present in the actions field, all legal OMA DM commands are allowed on the designated nodes(s) by the owner of the permission. Action names are interpreted case-insensitively, but the canonical action string returned by getActions() uses the forms defined by the action constants.

117.18.3.1 public static final String ADD = "Add"

Holders of DmtPermission with the Add action present can create new nodes in the DMT, that is they are authorized to execute the createInteriorNode() and createLeafNode() methods of the DmtSession. This action is also required for the copy() command, which needs to perform node creation operations (among others).

117.18.3.2 public static final String DELETE = "Delete"

Holders of DmtPermission with the Delete action present can delete nodes from the DMT, that is they are authorized to execute the deleteNode() method of the DmtSession.

117.18.3.3 public static final String EXEC = "Exec"

Holders of DmtPermission with the Exec action present can execute nodes in the DMT, that is they are authorized to call the execute() method of the DmtSession.

117.18.3.4 public static final String GET = "Get"

Holders of DmtPermission with the Get action present can query DMT node value or properties, that is they are authorized to execute the isLeafNode(), getNodeAcl(), getEffectiveNodeAcl(), get-

MetaNode(), getNodeValue(), getChildNodeNames(), getNodeTitle(), getNodeVersion(), getNodeTimeStamp(), getNodeSize() and getNodeType() methods of the DmtSession. This action is also required for the copy() command, which needs to perform node query operations (among others).

117.18.3.5 public static final String REPLACE = "Replace"

Holders of DmtPermission with the Replace action present can update DMT node value or properties, that is they are authorized to execute the setNodeAcl(), setNodeTitle(), setNodeValue(), setNodeType() and renameNode() methods of the DmtSession. This action is also be required for the copy() command if the original node had a title property (which must be set in the new node).

117.18.3.6 public DmtPermission(String dmtUri, String actions)

dmtUri URI of the management object (or subtree)

actions OMA DM actions allowed

□ Creates a new DmtPermission object for the specified DMT URI with the specified actions. The given URI can be:

- "*", which matches all valid (see Uri.isValidUri(String)) absolute URIs;
- the prefix of an absolute URI followed by the * character (for example "./OSGi/L*"), which matches all valid absolute URIs beginning with the given prefix;
- a valid absolute URI, which matches itself.

Since the * character is itself a valid URI character, it can appear as the last character of a valid absolute URI. To distinguish this case from using * as a wildcard, the * character at the end of the URI must be escaped with the \ character. For example the URI "./a*" matches "./a", "./aa", "./a/b" etc. while "./a*" matches "./a*" only.

The actions string must either be "*" to allow all actions, or it must contain a non-empty subset of the valid actions, defined as constants in this class.

Throws NullPointerException– if any of the parameters are null

IllegalArgumentException– if any of the parameters are invalid

117.18.3.7 public boolean equals(Object obj)

obj the object to compare to this DmtPermission instance

□ Checks whether the given object is equal to this DmtPermission instance. Two DmtPermission instances are equal if they have the same target string and the same action mask. The "*" action mask is considered equal to a mask containing all actions.

Returns true if the parameter represents the same permissions as this instance

117.18.3.8 public String getActions()

□ Returns the String representation of the action list. The allowed actions are listed in the following order: Add, Delete, Exec, Get, Replace. The wildcard character is not used in the returned string, even if the class was created using the "*" wildcard.

Returns canonical action list for this permission object

117.18.3.9 public int hashCode()

□ Returns the hash code for this permission object. If two DmtPermission objects are equal according to the equals(Object) method, then calling this method on each of the two DmtPermission objects must produce the same integer result.

Returns hash code for this permission object

117.18.3.10 public boolean implies(Permission p)

p the permission to check for implication

□ Checks if this DmtPermission object "implies" the specified permission. This method returns false if and only if at least one of the following conditions are fulfilled for the specified permission:

- it is not a DmtPermission
- its set of actions contains an action not allowed by this permission
- the set of nodes defined by its path contains a node not defined by the path of this permission

Returns true if this DmtPermission instance implies the specified permission

117.18.3.11 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object for storing DmtPermission objects.

Returns the new PermissionCollection

117.18.4 public class DmtPrincipalPermission extends Permission

Indicates the callers authority to create DMT sessions on behalf of a remote management server. Only protocol adapters communicating with management servers should be granted this permission.

DmtPrincipalPermission has a target string which controls the name of the principal on whose behalf the protocol adapter can act. A wildcard is allowed at the end of the target string, to allow using any principal name with the given prefix. The "*" target means the adapter can create a session in the name of any principal.

117.18.4.1 public DmtPrincipalPermission(String target)

target the name of the principal, can end with * to match any principal with the given prefix

□ Creates a new DmtPrincipalPermission object with its name set to the target string. Name must be non-null and non-empty.

Throws NullPointerException– if name is null

IllegalArgumentException– if name is empty

117.18.4.2 public DmtPrincipalPermission(String target, String actions)

target the name of the principal, can end with * to match any principal with the given prefix

actions no actions defined, must be "*" for forward compatibility

□ Creates a new DmtPrincipalPermission object using the 'canonical' two argument constructor. In this version this class does not define any actions, the second argument of this constructor must be "*" so that this class can later be extended in a backward compatible way.

Throws NullPointerException– if name or actions is null

IllegalArgumentException– if name is empty or actions is not "*"

117.18.4.3 public boolean equals(Object obj)

obj the object to compare to this DmtPrincipalPermission instance

□ Checks whether the given object is equal to this DmtPrincipalPermission instance. Two DmtPrincipalPermission instances are equal if they have the same target string.

Returns true if the parameter represents the same permissions as this instance

117.18.4.4 public String getActions()

□ Returns the action list (always * in the current version).

Returns the action string "*"

117.18.4.5 public int hashCode()

□ Returns the hash code for this permission object. If two DmtPrincipalPermission objects are equal according to the equals(Object) method, then calling this method on each of the two DmtPrincipalPermission objects must produce the same integer result.

Returns hash code for this permission object

117.18.4.6 public boolean implies(Permission p)

p the permission to check for implication

□ Checks if this DmtPrincipalPermission object implies the specified permission. Another DmtPrincipalPermission instance is implied by this permission either if the target strings are identical, or if this target can be made identical to the other target by replacing a trailing "*" with any string.

Returns true if this DmtPrincipalPermission instance implies the specified permission

117.18.4.7 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object for storing DmtPrincipalPermission objects.

Returns the new PermissionCollection

117.19 References

[1] *OMA DM-TND v1.2 draft*
https://www.openmobilealliance.org/release/DM/V1_2-20050607-C/OMA-TS-DM-TND-V1_2-20050607-C.pdf

[2] *OMA DM-RepPro v1.2 draft:*
https://www.openmobilealliance.org/release/DM/V1_2-20050607-C/OMA-TS-DM-RepPro-V1_2-20050607-C.pdf

[3] *IETF RFC2578. Structure of Management Information*
Version 2 (SMIv2)
https://www.ietf.org/rfc/rfc2578.txt

[4] *Java™ Management Extensions Instrumentation and Agent Specification v1.2, October 2002*,
https://www.oracle.com/java/technologies/javase/javamanagement.html

[5] *JSR 9 - Federated Management Architecture (FMA) Specification*
Version 1.0, January 2000
https://www.jcp.org/en/jsr/detail?id=9

[6] *WBEM Profile Template, DSP1000*
Status: Draft, Version 1.0 Preliminary, March 11, 2004
https://www.dmtf.org/standards/wbem

[7] *SNMP*
http://www.wtcs.org/snmp4tpc/snmp_rfc.htm#rfc

[8] *RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax*
https://www.ietf.org/rfc/rfc2396.txt

[9] *MIME Media Types*
https://www.iana.org/assignments/media-types/

[10] *RFC 3548 The Base16, Base32, and Base64 Data Encodings*
https://www.ietf.org/rfc/rfc3548.txt

[11] *Secure Hash Algorithm 1*
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[12] *TR-069 CPE WAN Management Protocol (CWMP)*
Customer Premises Equipment Wide Area Network Management Protocol (CWMP)
https://en.wikipedia.org/wiki/TR-069

[13] *XML Schema Part 2: Datatypes Second Edition*
https://www.w3.org/TR/xmlschema-2/

122 Remote Service Admin Service Specification

Version 1.1

122.1 Introduction

The *OSGi Core Release 8* framework specifies a model where bundles can use distributed services. The basic model for OSGi remote services is that a bundle can register services that are *exported* to a communication *Endpoint* and use services that are *imported* from a communication Endpoint. However, chapter *Remote Services* on page 25 does not explain *what* services are exported and/or imported; it leaves such decisions to the distribution provider. The distribution provider therefore performs multiple roles and cannot be leveraged by other bundles in scenarios that the distribution provider had not foreseen.

The primary role of the distribution provider is purely mechanical; it creates Endpoints and registers service proxies and enables their communication. The second role is about the policies around the desired topology. The third role is discovery. To establish a specific topology it is necessary to find out about exported services in other frameworks.

This specification therefore defines an API for the distribution provider and discovery of services in a network. A management agent can use this API to provide an actual distribution policy. This management agent, called the Topology Manager, can control the export and import of services delegating the intrinsic knowledge of the low level details of communication protocols, proxying of services, and discovering services in the network to services defined in this specification.

This specification is an extension of the Remote Service chapter. Though some aspects are repeated in this specification, a full understanding of the Remote Services chapter is required for full understanding of this document.

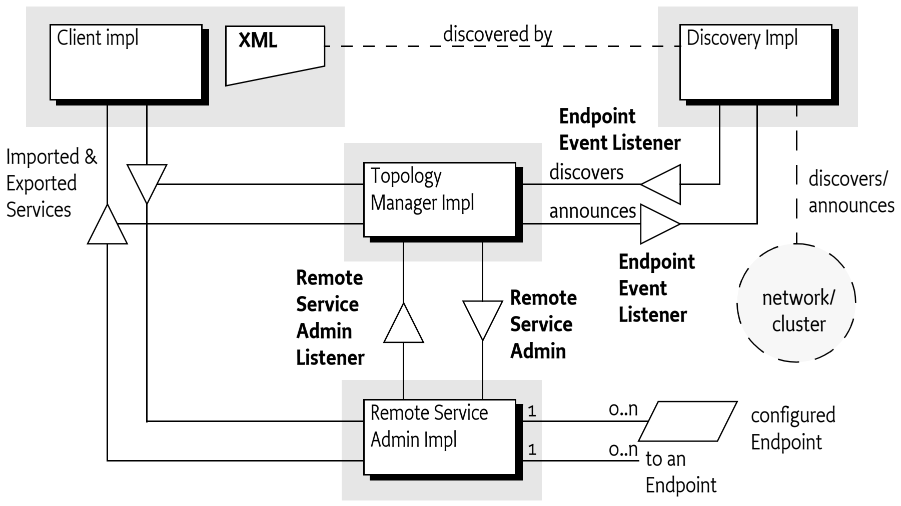
122.1.1 Essentials

- *Simple* - Make it as simple as possible for a Topology Manager to implement distribution policies.
- *Dynamic* - Discover available Endpoints dynamically, for example through a discovery protocol like [3] *Service Location Protocol (SLP)* or [4] *JGroups*.
- *Inform* - Provide a mechanism to inform other parties about created and removed Endpoints.
- *Configuration* - Allow bundles to describe Endpoints as a bundle resource that are provided to the Distribution Provider.
- *Selective* - Not all parties are interested in all services. Endpoint registries must be able to express the scope of services they are interested in.
- *Multiple* - Allow the collaboration of multiple Topology Managers, Remote Service Admin services, and Discovery Providers.
- *Dynamic* - Allow the dynamic discovery of Endpoints.
- *Federated* - Enable a global view of all available services in a distributed environment.

122.1.2 Entities

- *Remote Service Admin* - An implementation of this specification provides the mechanisms to import and export services through a set of configuration types. The Remote Service Admin service is a passive Distribution Provider, not taking any action to export or import itself.
- *Topology Manager* - The Topology Manager provides the policy for importing and exporting services through the Remote Service Admin service.
- *Endpoint* - An Endpoint is a communications access mechanism to a service in another framework, a (web) service, another process, or a queue or topic destination, etc., requiring some protocol for communications.
- *Endpoint Description* - A properties based description of an Endpoint. Endpoint Descriptions can be exchanged between different frameworks to create connections to each other's services. Endpoint Descriptions can also be created to Endpoints not originating in an OSGi Framework.
- *Endpoint Description Provider* - A party that can inform others about the existence of Endpoints.
- *Endpoint Event Listener* – A listener service that receives events relating to Endpoints that match its scope. This Endpoint Event Listener is used symmetrically to implement a federated registry. The Topology Manager can use it to notify interested parties about created and removed Endpoints, as well as to receive notifications from other parties, potentially remote, about their available Endpoints.
- *Endpoint Listener* – An older version of the Endpoint Event Listener defined by version 1.0 of this specification. The Endpoint Event Listener supersedes the Endpoint Listener, and should be used in preference where possible.
- *Remote Service Admin Listener* - A listener service that is informed of all the primitive actions that the Remote Service Admin performs like importing and exporting as well as errors.
- *Endpoint Configuration Extender* - A bundle that can detect configuration data describing an Endpoint Description in a bundle resource, using the extender pattern.
- *Discovery* – An Endpoint Event Listener that detects the Endpoint Descriptions through some discovery protocol.
- *Cluster* - A group of computing systems that closely work together, usually in a fast network.

Figure 122.1 Remote Service Admin Entities



122.1.3 Synopsis

Topology Managers are responsible for the distribution policies of a OSGi framework. To implement a policy, a Topology Manager must be aware of the environment, for this reason, it can register:

- Service listeners to detect services that can be exported according to the Remote Services chapter.
- Listener and Find Hook services to detect bundles that have an interest in specific services that potentially could be imported.
- A Remote Service Admin Listener service to detect the activity of other Topology Managers.
- Endpoint Event Listener and Endpoint Listener services to detect Endpoints that are made available through discovery protocols, configuration data, or other means.

Using this information, the manager implements a topology using the Remote Service Admin service. A Topology Manager that wants to export a service can create an *Export Registration* by providing one or more Remote Service Admin services a Service Reference plus a Map with the required properties. A Remote Service Admin service then creates a number of Endpoints based on the available configuration types and returns a collection of `ExportRegistration` objects. A collection is returned because a single service can be exported to multiple Endpoints depending on the available configuration type properties.

Each Export Registration is specific for the caller and represents an existing or newly created Endpoint. The Export Registration associates the exported Service Reference with an *Endpoint Description*. If there are problems with the export operation, the Remote Service Admin service reports these on the Export Registration objects. That is, not all the returned Export Registrations have to be valid.

An Endpoint Description is a property based description of an Endpoint. Some of these properties are defined in this specification, other properties are defined by configuration types. These configuration types must follow the same rules as the configuration types defined in the Remote Services chapter. Remote Service Admin services that support the configuration types in the Endpoint Description can import a service from that Endpoint solely based on that Endpoint Description.

In similar vein, the Topology Manager can import a service from a remote system by creating an Import Registration out of an Endpoint Description. The Remote Service Admin service then registers a service that is a proxy for the remote Endpoint and returns an `ImportRegistration` object. If there are problems with the import, the Remote Service Admin service that cannot be detected early, then the Remote Service Admin service reports these on the returned `ImportRegistration` object.

For introspection, the Remote Service Admin can list its current set of Import and Export References so that a Topology Manager can get the current state. The Remote Service Admin service also informs all Topology Managers and observers of the creation, deletion, and errors of Import and Export Registrations through the Remote Service Admin Listener service. Interested parties like the Topology Manager can register such a service and will be called back with the initial state as well as any subsequent changes.

An important aspect of the Topology Manager is the distributed nature of the scenarios it plays an orchestrating role in. A Topology Manager needs to be aware of Endpoints in the network, not just the ones provided by Remote Service Admin services in its local framework. The Endpoint Event Listener service is specified for this purpose. This service is provided for both directions, symmetrically. That is, it is used by the Topology Manager to inform any observers about the existence of Endpoints that are locally available, as well as for parties that represent a discovery mechanism. For example Endpoints available on other systems, Endpoint Descriptions embedded in resources in bundles, or Endpoint Descriptions that are available in some other form.

Endpoint Event Listener services are not always interested in the complete set of available Endpoints because this set can potentially be very large. For example, if a remote registry like [5] *UDDI* is used then the number of Endpoints can run into the thousands or more. An Endpoint Event Listener service can therefore scope the set of Endpoints with an OSGi LDAP style filter. Parties that can provide information about Endpoints must only notify Endpoint Event Listener services when the Endpoint Description falls within the scope of the Endpoint Listener service. Parties that use some discovery mechanism can use the scope to trigger directed searches across the network.

122.1.3.1 Endpoint Listener Services

The 1.0 version of this specification defined an Endpoint Listener service, which has an identical purpose and similar behaviors to an Endpoint Event Listener service. Unfortunately the design of the Endpoint Listener limited its extensibility, meaning that it had to be replaced in version 1.1 of this specification.

In order to maintain backward compatible interoperability with Remote Service Admin 1.0 actors, Remote Service Admin 1.1 actors must continue to register Endpoint Listener services as well as Endpoint Event Listener services. They must also continue to call Endpoint Listener services as well as EndpointEventListener services.

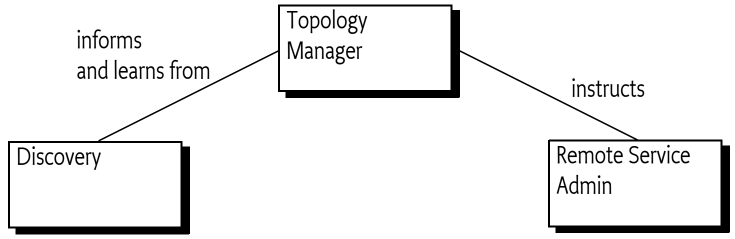
122.2 Actors

The OSGi Remote Services specification is about the distribution of services. This specification does not outline the details of how the distribution provider knows the desired topology, this policy aspect is left up to implementations. In many situations, this is a desirable architecture because it provides freedom of implementation to the distribution provider. However, such an architecture does not enable a separation of the mechanisms and *policy*. Therefore, this Remote Service Admin specification provides an architecture that enables a separate bundle from the distribution provider to define the topology. It splits the responsibility of the Remote Service specification in a number of *roles*. These roles can all have different implementations but they can collaborate through the services defined in this specification. These roles are:

- *Topology Manager*s - Topology Managers are the (anonymous) players that implement the policies for distributing services; they are closely aligned with the concept of an OSGi *management agent*. It is expected that Topology Managers will be developed for scenarios like import/export all applicable services, configuration based imports- and exports, and scenarios like fail-over, load-balancing, as well as standards like domain managers for the [6] *Service Component Architecture (SCA)*.
- *Remote Service Admin* - The Remote Service Admin service provides the basic mechanism to import and export services. This service is policy free; it will not distribute services without explicitly being told so. A OSGi framework can host multiple Remote Service Admin services that, for example, support different configuration types.
- *Discovery* - To implement a distribution policy, a Topology Manager must be aware of what Endpoints are available. This specification provides an abstraction of a *federated Endpoint registry*. This registry can be used to both publish as well as consume Endpoints from many different sources. The federated registry is defined for local services but is intended to be used with standard and proprietary service discovery protocols. The federated registry is implemented with the Endpoint Event Listener service.

These roles are depicted in Figure 122.2 on page 496.

Figure 122.2 Roles



122.3 Topology Managers

Distributed processing has become mainstream because of the massive scale required for Internet applications. Only with distributed architectures is it possible to scale systems to *Internet size* with hundreds of millions of users. To allow a system to scale, servers are grouped in clusters where they can work in unison or geographically dispersed in even larger configurations. The distribution of the work-load is crucial for the amount of scalability provided by an architecture and often has domain specific dispatching techniques. For example, the hash of a user id can be used to select the correct profile database server. In this fast moving world it is very unlikely that a single architecture or distribution policy would be sufficient to satisfy many users. It is therefore that this specification separates the *how* from the *what*. The complex mechanics of importing and exporting services are managed by a Remote Service Admin service (the how) while the different policies are implemented by Topology Managers (the what). This separation of concerns enables the development of Topology Managers that can run on many different systems, providing high user functionality. For example, a Topology Manager could implement a fail-over policy where some strategic services are redirected when their connections fail. Other Topology Managers could use a discovery protocol like SLP to find out about other systems in a cluster and automatically configure the cluster.

The key value of this architecture is demonstrated by the example of an *SCA domain controller*. An SCA domain controller receives a description of a domain (a set of systems and modules) and must ensure that the proper connections are made between the participating SCA modules. By splitting the roles, an SCA domain manager can be developed that can run on any compatible Remote Service Admin service implementation.

122.3.1 Multiple Topology Managers

There is no restriction on the number of Topology Managers, nor is there a restriction on the number of Remote Service Admin service implementations. It is up to the deployer of the OSGi framework to select the appropriate set of these service implementations. It is the responsibility of the Topology Managers to listen to the Remote Service Admin Listener and track Endpoints created and deleted by other Topology Managers and act appropriately.

122.3.2 Example Use Cases

122.3.2.1 Promiscuous Policy

A *cluster* is a set of machines that are connected in a network. The simplest policy for a Topology Manager is to share exported services in such a cluster. Such a policy is very easy to implement with the Remote Services Admin service. In the most basic form, this Topology Manager would use some multicast protocol to communicate with its peers. These peers would exchange `EndpointDescription` objects of exported services. Each Topology Manager would then import any exported service.

This scenario can be improved by separating the promiscuous policy from the discovery. Instead of embedding the multicast protocol, a Topology manager could use the Endpoint Event Listener service. This service allows the discovery of remote services. At the same time, the Topology Manager could tell all other Endpoint Event Listener services about the services it has created, allowing them to be used by others in the network.

Splitting the Topology Manager and discovery in two bundles allows different implementations of the discovery bundle, for example, to use different protocols. See `PROMISCUOUS_POLICY`.

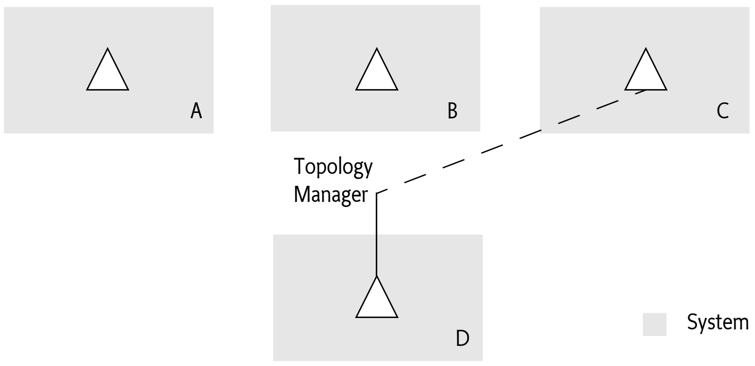
122.3.2.2 Fail Over

A more elaborate scheme is a *fail-over policy*. In such a policy a service can be replaced by a service from another machine. There are many ways to implement such a policy, an simple example strategy is provided here for illustration.

A Fail-Over Topology Manager is given a list of stateless services that require fail-over, for example through the *Configuration Admin Service Specification* on page 65. The Fail-Over Manager tracks the systems in the its cluster that provide such services. This tracking can use an embedded protocol or it can be based on the Endpoint Event Listener service model.

In the Fail-Over policy, the fail-over manager only imports a single service and then tracks the error status of the imported service through the Remote Service Admin Listener service. If it detects the service is becoming unavailable, it closes the corresponding Import Registration and imports a service from an alternative system instead. In Figure 122.3, there are 4 systems in a cluster. The topology/fail-over manager ensures that there is always one of the services in system A, B, or C available in D.

Figure 122.3 *Fail Over Scenario in a cluster*



There are many possible variations on this scenario. The managers could exchange load information, allowing the service switch to be influenced by the load of the target systems. The important aspect is that the Topology Manager can ignore the complex details of discovery protocols, communication protocols, and service proxying and instead focus on the topology. See FAIL_OVER_POLICY.

122.4 Endpoint Description

An *Endpoint* is a point of rendezvous of distribution providers. It is created by an exporting distribution provider or some other party, and is used by importing distribution providers to create a connection. An *Endpoint Description* describes an Endpoint in such a way that an importing Remote Service Admin service can create this connection if it recognizes the *configuration type* that is used for that Endpoint. The configuration type consists of a name and a set of properties associated with that name.

The core concept of the Endpoint Description is a Map of properties. The structure of this map is the same as service properties, and the defined properties are closely aligned with the properties of an imported service. An EndpointDescription object must only consist of the data types that are supported for service properties. This makes the property map serializable with many different mechanisms. The EndpointDescription class provides a convenient way to access the properties in a type safe way.

An Endpoint Description has case insensitive keys, just like the Service Reference's properties.

The properties map must contain all the prescribed service properties of the exported service after intents have been processed, as if the service was registered as an imported service. That is, the map must not contain any properties that start with service.exported.* but it must contain the service.imported.* variation of these properties. The Endpoint Description must reflect the imported service properties because this simplifies the use of filters from the service hooks. Filters applied to the Endpoint Description can then be the same filters as applied by a bundle to select an imported service from the service registry.

The properties that can be used in an Endpoint Description are listed in Table 122.1. The RemoteConstants class contains the constants for all of these property names.

Table 122.1 Endpoint Properties

Endpoint Property Name	Type	Description
service.exported.*		Must not be set
service.imported	*	Must always be set to some value. See SERVICE_IMPORTED.
objectClass	String[]	Must be set to the value of service.exported.interfaces, of the exported service after expanding any wildcards. Though this property will be overridden by the framework for the corresponding service registration, it must be set in the Endpoint Description to simplify the filter matching. These interface names are available with the getInterfaces() method.
service.intents	String+	Intents implemented by the exporting distribution provider and, if applicable, the exported service itself. Any qualified intents must have their expanded form present. These expanded intents are available with the getIntents() method. See SERVICE_INTENTS.
endpoint.service.id	Long	The service id of the exported service. Can be absent or 0 if the corresponding Endpoint is not for an OSGi service. The remote service id is available as getServiceId(). See also ENDPOINT_SERVICE_ID.
endpoint.framework.uuid	String	A universally unique id identifying the instance of the exporting framework. Can be absent if the corresponding Endpoint is not for an OSGi service. See *Framework UUID* on page 501. The remote framework UUID is available with the getFrameworkUUID() method. See also ENDPOINT_FRAMEWORK_UUID.
endpoint.id	String	The Id for this Endpoint, can never be null. This information is available with the getId(). See *Endpoint Id* on page 501 and also ENDPOINT_ID.
endpoint.package. version.<package-name>	String	The Java package version for the embedded <package>. For example, the property endpoint.package.version.com.acme=1.3 describes the version for the com.acme package. The version for a package can be obtained with the getPackageVersion(String). The version does not have to be set, if not set, the value must be assumed to be 0.
service.imported.configs	String+	The configuration types that can be used to implement the corresponding Endpoint. This property maps to the corresponding property in the Remote Services chapter. This property can be obtained with the getConfigurationTypes() method. The Export Registration has all the possible configuration types, where the Import Registration reports the configuration type actually used. SERVICE_IMPORTED_CONFIGS.

Endpoint Property Name	Type	Description
`<config>.*`	*	Where `<config>` is one of the configuration type names listed in `service.imported.configs`. The content of these properties must be valid for creating a connection to the Endpoint in another framework. That is, any locally readable URLs from bundles must be converted in such a form that they can be read by the importing framework. How this is done is configuration type specific.
*	*	All remaining public service properties must be present (that is, not starting with full stop ('.' `\u002E`)). If the values can not be marshaled by the Distribution Provider then they must be ignored.

The `EndpointDescription` class has a number of constructors that make it convenient to instantiate it for different purposes:

- `EndpointDescription(Map)` - Instantiate the Endpoint Description from a `Map` object.
- `EndpointDescription(ServiceReference,Map)` - Instantiate an Endpoint Description based on a Service Reference and a Map. The base properties of this Endpoint Description are the Service Reference properties but the properties in the given Map must override any of their case variants in the Service Reference. This allows the construction of an Endpoint Description from an exportable service while still allowing overrides of specific properties by the Topology Manager.

The Endpoint Description must use the allowed properties as given in Table 122.1 on page 499. The Endpoint Description must automatically skip any `service.exported.*` properties.

The Endpoint Description provides the following methods to access the properties in a more convenient way:

- `getInterfaces()` - Answers a list of Java interface names. These are the interfaces under which the services must be registered. These interface names can also be found at the `objectClass` property. A service can only be imported when there is at least one Java interface name available.
- `getConfigurationTypes()` - Answer the configuration types that are used for exporting this Endpoint. The configuration types are associated with a number of properties.
- `getId()` - Returns an Id uniquely identifying an Endpoint. The syntax of this Id should be defined in the specification for the associated configuration type. Two Endpoint Descriptions with the same Id describe the same Endpoint.
- `getFrameworkUUID()` - Get a Universally Unique Identifier (UUID) for the framework instance that has created the Endpoint, *Framework UUID* on page 501.
- `getServiceId()` - Get the service id for the framework instance that has created the Endpoint. If there is no service on the remote side the value must be 0.
- `getPackageVersion(String)` - Get the version for the given package.
- `getIntents()` - Get the list of specified intents.
- `getProperties()` - Get all the properties.

Two Endpoint Descriptions are deemed equal when their Endpoint Id is equal. The Endpoint Id is a mandatory property of an Endpoint Description, it is further described at *Endpoint Id* on page 501. The hash code is therefore also based on the Endpoint Id.

122.4.1 Validity

A valid Endpoint Description must at least satisfy the following assertions:

- It must have a non-`null` Id that uniquely identifies the Endpoint

- It must at least have one Java interface name
- It must at least have one configuration type set
- Any version for the packages must have a valid version syntax.

122.4.2 Mutability

An `EndpointDescription` object is immutable and with all final fields. It can be freely used between different threads.

122.4.3 Endpoint Id

An Endpoint Id is an opaque unique identifier for an Endpoint. This uniqueness must at least hold for the entire network in which the Endpoint is used. There is no syntax defined for this string except that white space at the beginning and ending must be ignored. The actual syntax for this Endpoint Id must be defined by the actual configuration type.

Two Endpoint Descriptions are deemed identical when their Endpoint Id is equal. The Endpoint Ids must be compared as string compares with leading and trailing spaces removed. The Endpoint Description class must use the `String` class' hash Code from the Endpoint Id as its own `hashCode`.

The simplest way to ensure that a growth in the number of EndpointDescriptions and/or the size of the connected group does not violate the required uniqueness of Endpoint Ids is for implementations to make their Endpoint Ids globally unique. This protects against clashes regardless of changes to the connected group.

Whilst globally unique identifiers (GUIDs) are a simple solution to the Endpoint Id uniqueness problem, they are not easy to implement in all environments. In some systems they can be prohibitively expensive to create, or of insufficient entropy to be genuinely unique. Some distribution providers may therefore choose not to use random GUIDs.

In the case where no globally unique value is used the following actions are recommended (although not required).

- Distribution Providers protect against intra-framework clashes using some known value unique to the service, for example the service id.
- Distribution Providers protect against inter-provider collisions within a single framework by using some unique value, such as the distribution provider's bundle id. The distribution provider bundle's symbolic name is insufficient, as there may be multiple versions of the same distribution provider installed within a single framework.
- Distribution Providers protect against inter-framework collisions using some value unique to the framework, such as the framework UUID.

122.4.4 Framework UUID

Each framework registers its services with a service id that is only unique for that specific framework. The OSGi framework is not a singleton, making it possible that a single VM process holds multiple OSGi frameworks. Therefore, to identify an OSGi service uniquely it is necessary to identify the framework that has registered it. This identifier is a *Universally Unique IDentifier* (UUID) that is set for each framework. This UUID is contained in the following framework property:

```
org.osgi.framework.uuid
```

If an Endpoint Description has no associated OSGi service then the UUID of that Endpoint Description must not be set and its service id must be 0.

A local Endpoint Description will have its framework UUID set to the local framework. This makes it straightforward to filter for Endpoint Descriptions that are describing local Endpoints or that describe remote Endpoints. For example, a manager can take the filter from a listener and ensure that it is only getting remote Endpoint Descriptions:

```
(&
  (!
    (service.remote.framework.uuid
       =72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
  )
  (objectClass=org.osgi.service.log.LogService)
)
```

Where `72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72` is the UUID of the local framework. A discovery bundle can register the following filter in its scope to receive all locally generated Endpoints:

```
(service.remote.framework.uuid
       =72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
```

122.4.5 Resource Containment

Configuration types can use URLs to point to local resources describing in detail the Endpoint parameters for specific protocols. However, the purpose of an Endpoint Description is to describe an Endpoint to a remote system. This implies that there is some marshaling process that will transfer the Endpoint Description to another process. This other process is unlikely to be able to access resource URLs. Local bundle resource URLs are only usable in the framework that originates them but even HTTP based URLs can easily run into problems due to firewalls or lack of routing.

Therefore, the properties for a configuration type should be stored in such a way that the receiving process can access them. One way to achieve this is to contain the configuration properties completely in the Endpoint Description and ensure they only use the basic data types that the remote services chapter in the core requires every Distribution Provider to support.

The Endpoint Description XML format provides an `xml` element that is specifically added to make it easy to embed XML based configuration documents. The XML Schema is defined in *Endpoint Description Extender Format* on page 513.

122.5 Remote Service Admin

The Remote Service Admin service abstracts the core functionality of a distribution provider: exporting a service to an Endpoint and importing services from an Endpoint. However, in contrast with the distribution provider of the Remote Services specification, the Remote Service Admin service must be told explicitly what services to import and export.

122.5.1 Exporting

An exportable service can be exported with the `exportService(ServiceReference,Map)` method. This method creates a number of Endpoints by inspecting the merged properties from the Service Reference and the given Map. Any property in the Map overrides the Service Reference properties, regardless of case. That is, if the map contains a key then it will override any case variant of this key in the Service Reference. However, if the Map contains the `objectClass` or `service.id` property key in any case variant, then these properties must not override the Service Reference's value.

The Remote Service Admin service must interpret the merged properties according to the Remote Services chapter. This means that it must look at the following properties (as defined in chapter *Remote Services* on page 25):

- `service.exported.configs` - (`String+`) A list of configuration types that should be used to export this service. Each configuration type represents the configuration parameters for an Endpoint. A Remote Service Admin service should create an Endpoint for each configuration type that it supports and ignore the types it does not recognize. If this property is not set, then the Remote

Service Admin implementation must choose a convenient configuration type that then must be reported on the Endpoint Description with the service.imported.configs associated with the returned Export Registration.

- service.exported.intents - (String+) A list of intents that the Remote Service Admin service must implement to distribute the given service.
- service.exported.intents.extra - (String+) This property is merged with the service.exported.intents property.
- service.exported.interfaces - (String+) This property must be set; it marks this service for export and defines the interfaces. The list members must all be contained in the types listed in the objectClass service property from the Service Reference. The single value of an asterisk ('*' \u002A) indicates all interfaces in the registration's objectClass property and ignore the classes. Being able to set this property outside the Service Reference implies that the Topology Manager can export any registered service, also services not specifically marked to be exported.
- service.intents - (String+) A list of intents that this service has implemented.

A Topology Manager cannot remove properties, null is invalid as a property value.

The Remote Service Admin returns a collection of ExportRegistration objects. This collection must contain an entry for each configuration type the Remote Service Admin has recognized. Unrecognized configuration types must be ignored. Recognized configuration types which require intents that are not supported by the Remote Service Admin must also be ignored. However, it is possible that this list contains *invalid registrations*, see *Invalid Registrations* on page 506.

If a Service was already exported then the Remote Service Admin must still return a new ExportRegistration object that is linked with the earlier registrations. That is, an Endpoint can be shared between multiple Export Registrations. The Remote Service Admin service must ensure that the corresponding Endpoint remains available as long as there is at least one open Export Registration for that Endpoint.

For each successful creation of an export registration, the Remote Service Admin service must publish an EXPORT_REGISTRATION event, see *Events* on page 511. This event must be emitted, even if the Endpoint already existed and is thus shared with another Export Registration. If the creation of an Endpoint runs into an error, an EXPORT_ERROR event must be emitted.

Each valid Export Registration corresponds to an Endpoint for the given service. This Endpoint must remain active until all of the Export Registrations are closed that share this Endpoint.

The Endpoint can now be published so that other processes or systems can import this Endpoint. To aid with this import, the Export Registration has a getExportReference() method that returns an ExportReference object. This reference provides the following information:

- getExportedEndpoint() - This is the associated Endpoint Description. This Endpoint Description is a properties based description of an Endpoint. The property keys and their semantics are outlined in *Endpoint Description* on page 498. It can be used to inform other systems of the availability of an Endpoint.
- getExportedService() - The Service Reference to the exported service.

Both methods must return null when the associated Export Registration is closed.

A Distribution Provider that recognizes the configuration type in an Endpoint can create a connection to an Endpoint on other systems as long as firewalls and networks permit. The Endpoint Description can therefore be communicated to other systems to announce the availability of an Endpoint. The Topology Manager can optionally announce the availability of an Endpoint to the Endpoint Event Listener services, see *Discovery* on page 507. The decision to announce the availability of an Endpoint is one of the policies that is provided by a specific Topology Manager.

The Export Registrations remain open until:

- Explicitly closed by the Topology Manager, or

- The Remote Service Admin service is no longer used by the Topology Manager that created the Export Registration.

If the Remote Service Admin service can no longer maintain the corresponding Endpoint due to failures than these should be reported through the events. However, the registrations should remain open until explicitly closed by the Topology Manager.

See *Registration Life Cycle* on page 506 for more information.

The Export Registrations are not permanent; persistence is in the realm of the Topology Manager.

122.5.2 Importing

To import a service, a Topology Manager must have an Endpoint Description that describes the Endpoint the imported service should connect to. With this Endpoint Description, a Remote Service Admin service can then import the corresponding Endpoint. A Topology Manager can obtain these Endpoint Descriptions through internal configuration; it can use the discovery model enabled by the Endpoint Event Listener service, see *Discovery* on page 507, or some alternate means.

A service can be imported with the Remote Service Admin importService(EndpointDescription) method. This method takes an Endpoint Description and picks one of the embedded configuration types to establish a connection with the corresponding Endpoint to create a local service proxy. This proxy can then be mapped to either a remote OSGi service or an alternative, for example a web service. In certain cases the service proxy can be lazy, only verifying the reachability of the Endpoint when it is actually invoked for the first time. This implies that a service proxy can block when invoked until the proper communication setup has taken place.

If the Remote Service Admin service does not recognize any of the configuration types then it must return null. If there are multiple configuration types recognized then the Remote Service Admin is free to select any one of the recognized types.

The Remote Service Admin service must ensure that service properties are according to the Remote Services chapter for an imported service. This means that it must register the following properties:

- service.imported - (*) Must be set to any value.
- service.imported.configs - (String+) The configuration information used to import this service. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework, see *Resource Containment* on page 502. Multiple configuration types can be listed if they are synonyms for exactly the same Endpoint that is used to export this service.
- service.intents - (String+) The Remote Service Admin must set this property to convey the combined intents of:
  - The exporting service, and
  - The intents that the exporting distribution provider adds, and
  - The intents that the importing distribution provider adds.
- Any additional properties listed in the Endpoint Description that should not be excluded. See *Endpoint Description* on page 498 for more details about the properties in the Endpoint Description.

A Remote Service Admin service must strictly follow the rules for importing a service as outlined in the Remote Services chapter.

The Remote Service Admin must return an ImportRegistration object or null. Even if an Import Registration is returned, it can still be an *invalid registration*, see *Invalid Registrations* on page 506 if the setup of the connection failed asynchronously. The Import Registration must always be a new object. Each valid Import Registration corresponds to a proxy service, potentially shared, that was created for the given Endpoint. The issues around proxying are described in *Proxying* on page 506.

For each successful creation of an import registration, the Remote Service Admin service must publish an IMPORT_REGISTRATION event, if there is an error it must publish an IMPORT_ERROR, see *Events* on page 511.

For more information see *Registration Life Cycle* on page 506.

The Import Registration provides access to an ImportReference object with the getImportReference(). This object has the following methods:

- getImportedEndpoint() - Provides the Endpoint Description for this imported service.
- getImportedService() - Provides the Service Reference for the service proxy.

The Import Registration will remain open as long as:

- The corresponding remote Endpoint remains available, and
- The Remote Service Admin service is still in use by the Topology Manager that created the Import Registration.

That is, the Import Registrations are not permanent, any persistence is in the realm of the Topology Manager. See *Registration Life Cycle* on page 506 for more details.

122.5.3 Updates

Services Registrations are dynamic and service properties may change during the lifetime of a service. Remote services must mirror these dynamics without making it appear as though the service has become unavailable. This requires that the exporting distribution provider and the importing distribution provider support the changing of service properties.

There are two types of service properties:

- Properties that are intended to be consumed by the distribution provider, such as: the exported interfaces and configuration types, exported intents and configuration type specific properties. These properties are typically prefixed with 'service.' or 'endpoint.' see Table 122.1 on page 499.
- Service properties not intended for the distribution provider. These are typically used to communicate information to the consumer of the service and are often specific to the domain of the service.

The following methods to support the updating of service properties on Export Registrations and the propagation of these updates to the remote proxies via Import Registrations.

- ExportRegistration.update(Map) - Allows the Topology Manager to update an existing export registration it created after receiving a notification of changed properties on the remoted service.
- ImportRegistration.update(EndpointDescription) - Allows the Topology Manager to update the import registration representing a remote service after the remote service properties have been updated. Typically the topology manager is notified of such change via the Discovery mechanism.

The distribution provider must support the updates of service properties *not* intended for the distribution provider, where supported property values are as defined in the *Filter Syntax* of *OSGi Core Release 8*. Distribution providers may support updates to a wider set of properties or data types, but these may fail with other implementations.

122.5.4 Reflection

The Remote Service Admin service provides the following methods to get the list of the current exported and imported services:

- getExportedServices() - List the Export References for services that are exported by this Remote Service Admin service as directed by any of the Topology Managers.
- getImportedEndpoints() - List the Import References for services that have been imported by this Remote Service Admin service as directed by any of the Topology Managers.

122.5.5 Registration Life Cycle

All registrations obtained through a Remote Service Admin service are life cycle bound to the Topology Manager that created it. That is, if a Topology Manager ungets its Remote Service Admin service, all registrations obtained through this service must automatically be closed. This model ensures that all registrations are properly closed if either the Remote Service Admin or the Topology Manager stops because in both cases the framework performs the unget automatically. Such behavior can be achieved by implementing the Remote Service Admin service as a Service Factory.

122.5.6 Invalid Registrations

The Remote Service Admin service is explicitly allowed to return *invalid* Import and Export Registrations. First, in a communications stack it can take time to discover that there are issues, allowing the registration to return before it has completed can potentially save time. Second, it allows the Topology Manager to discover problems with the configuration information. Without the invalid Export Registrations, the Topology Manager would have to scan the log or associate the Remote Service Admin Events with a specific import/export method call, something that can be difficult to do.

If the registration is invalid, the getException() method must return a Throwable object. If the registration has initialized correctly, this method will return null. The getExportReference() and getImportReference() methods must throw an Illegal State Exception when the registration is invalid. A Remote Service Admin service is allowed to block for a reasonable amount of time when any of these methods is called, including the getException method, to finish initialization.

An invalid registration can be considered as never having been opened, it is therefore not necessary to close it; however, closing an invalid or closed registration must be a dummy operation and never throw an Exception. However, a failed registration must generate a corresponding error event.

122.5.7 Proxying

It is the responsibility of the Remote Service Admin service to properly proxy an imported service. This specification does not mandate the technique used to proxy an Endpoint as a service in the OSGi framework. The OSGi Remote Services specification allows a distribution provider to limit what it can proxy.

One of the primary aspects of a proxy is to ensure class space consistency between the exporting bundle and importing bundles. This can require the generation of a proxy-per-bundle to match the proper class spaces. It is the responsibility of the Remote Service Admin to ensure that no Class Cast Exceptions occur.

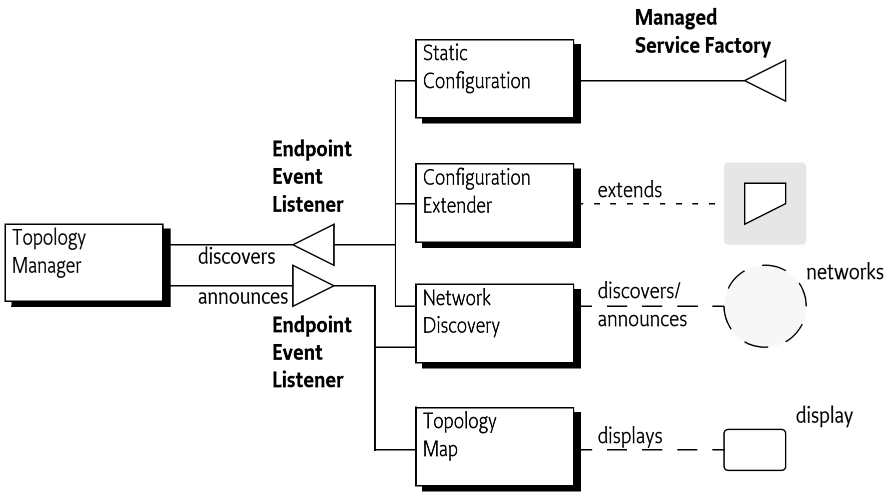
A common technique to achieve maximum class space compatibility is to use a Service Factory. A Service Factory provides the calling bundle when it first gets the service, making it straightforward to verify the package version of the interface that the calling bundle uses. Knowing the bundle that requests the service allows the creation of specialized proxies for each bundle. The interface class(es) for the proxy can then be loaded directly from the bundle, ensuring class compatibility. Interfaces should be loadable by the bundle otherwise that bundle can not use the interface in its code. If an interface cannot be loaded then it can be skipped. A dedicated class loader can then be created that has visibility to all these interfaces and is used to define the proxy class. This design ensures proper visibility and consistency. Implementations can optimize this model by sharing compatible class loaders between bundles.

The proxy will have to call arbitrary methods on arbitrary services. This has a large number of security implications, see *Security* on page 521.

122.6 Discovery

The topology of the distributed system is decided by the Topology Manager. However, in a distributed environment, the Topology Manager needs to *discover* Endpoints in other frameworks. There is a very large number of ways how a Topology Manager could learn about other Endpoints, ranging from static configuration, a centralized administration, all the way to fully dynamic discovery protocols like the Service Location Protocol (SLP) or JGroups. To support the required flexibility, this specification defines an *Endpoint Event Listener* service that allows the dissemination of Endpoint information. This service provides a symmetric solution because the problem is symmetric: it is used by a Topology Manager to announce changes in its local topology as well as find out about other Endpoint Descriptions. Where those other Endpoint Descriptions come from can vary widely. This design is depicted in Figure 122.4 on page 507.

Figure 122.4 *Examples*



The design of the Endpoint Event Listener allows a federated registry of Endpoint Descriptions. Any party that is interested in Endpoint Descriptions should register an Endpoint Event Listener service. This will signal that it is interested in topology information to any *Endpoint Description Providers*. Each Endpoint Event Listener service must be registered with a service property that holds a set of filter strings to indicate the *scope* of its interest. These filters must match an Endpoint Description before the corresponding Endpoint Event Listener service is notified of the availability of an Endpoint Description. Scoping is intended to limit the delivery of unnecessary Endpoint Descriptions as well as signal the need for specific Endpoints.

In addition to providing an Endpoint Event Listener actors must provide an Endpoint Listener. This may, or may not, be the same service object as the Endpoint Event Listener. Registering an Endpoint Listener in addition to an Endpoint Event Listener ensures that Endpoint announcements from version 1.0 actors will continue to be visible. If a service object is advertised as both an Endpoint Listener *and* an Endpoint Event Listener then version 1.1 actors must use the Endpoint Event Listener interface of the service in preference, and not call it as an Endpoint Listener. For this reason the Endpoint Listener interface is marked as `Deprecated`. The reason that the Endpoint Event Listener interface should be preferred is that it supports more advanced notification types, such as modification events.

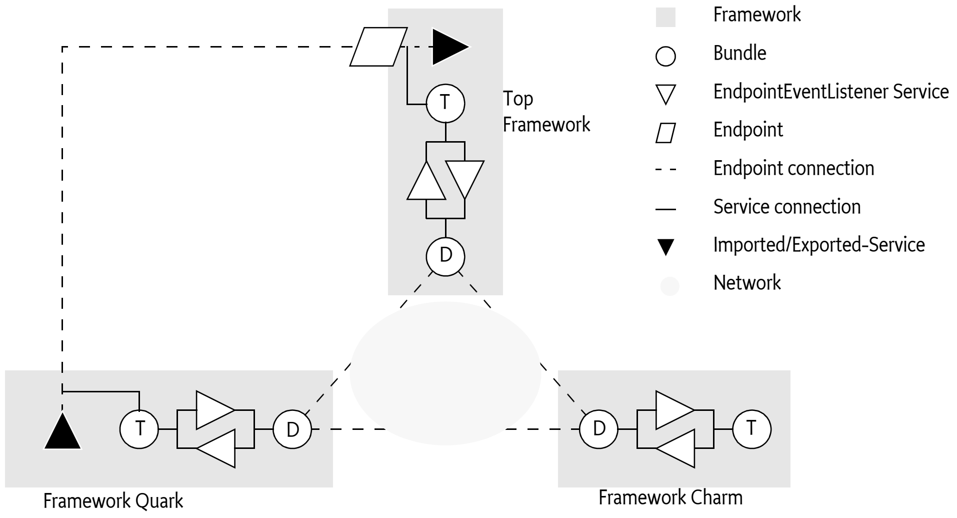
A Topology Manager has knowledge of its local Endpoints and is likely to be only interested in remote Endpoints. It can therefore set the scope to only match remote Endpoint Descriptions. See *Framework UUID* on page 501 for how to limit the scope to local or remote Endpoints. At the same time, a Topology manager should inform any locally registered Endpoint Event Listener and Endpoint Listener services about Endpoints that it has created or deleted.

This architecture allows many different use cases. For example, a bundle could display a map of the topology by registering an Endpoint Event Listener with a scope for local Endpoints. Another example is the use of SLP to announce local Endpoints to a network and to discover remote Endpoints from other parties on this network.

An instance of this design is shown in Figure 122.5 on page 508. In this figure, there are 3 frameworks that collaborate through some discovery bundle. The Top framework has created an Endpoint and decides to notify all Endpoint Event Listeners and Endpoint Listeners registered in this framework that are scoped to this new Endpoint. Local bundle D has set its scope to all Endpoint Descriptions that originate from its local framework, it therefore receives the Endpoint Description from T. Bundle D then sends the Endpoint Description to all its peers on the network.

In the Quark framework, the manager bundle T has expressed an interest by setting its scope to a filter that matches the Endpoint Description from the Top framework. When the bundle D on the Quark framework receives the Endpoint Description from bundle D on the Top framework, it matches it against all local Endpoint Event Listener's scope. In this case, the local manager bundle T matches and is given the Endpoint Description. The manager then uses the Remote Service Admin service to import the exported service described by the given Endpoint Description.

Figure 122.5 Endpoint Discovery Architecture. T=Topology Manager, D=Discovery



The previous description is just one of the possible usages of the Endpoint Event Listener. For example, the discovery bundles could communicate the scopes to their peers. These peers could then register an Endpoint Event Listener per peer, minimizing the network traffic because Endpoint Descriptions do not have to be broadcast to all peers.

Another alternative usage is described in *Endpoint Description Extender Format* on page 513. In this chapter the extender pattern is used to retrieve Endpoint Descriptions from resources in locally active bundles.

122.6.1 Scope and Filters

An Endpoint Event Listener or Endpoint Listener service is registered with the ENDPOINT_LISTENER_SCOPE service property. This property, which is String+, must be set and must contain at least one filter. If there is not at least one filter, then that Endpoint Event Listener or Endpoint Listener must not receive any Endpoint Descriptions.

Each filter in the scope is applied against the properties of the Endpoint Description until one succeeds. Only if one succeeds is the Endpoint informed about the existence of an Endpoint.

The Endpoint Description is designed to reflect the properties of the imported service, there is therefore a correspondence with the filters that are used by bundles that are listening for service registrations. The purpose of this design is to match the filter available through Listener Hook services, see *On Demand* on page 511.

However, the purpose of the filters is more generic than just this use case. It can also be used to specify the interest in local Endpoints or remote Endpoints. For example, Topology Managers are only interested in remote Endpoints while discoverers are only interested in local Endpoints. It is easy to discriminate between local and remote by filtering on the `endpoint.framework.uuid` property. Endpoint Descriptions contain the Universally Unique ID (UUID) of the originating framework. This UUID must be available from the local framework as well. See *Framework UUID* on page 501.

122.6.2 Endpoint Event Listener Interface

The `EndpointEventListener` interface has the following method:

- `endpointChanged(EndpointEvent,String)` – Notify the Endpoint Event Listener of changes to an Endpoint. The change could entail the addition or removal of an Endpoint or the modification of the properties of an existing Endpoint. Multiple identical events should be counted as a single such event.

These methods must only be called if the Endpoint Event Listener service has a filter in its scope that matches the Endpoint Description properties.

The Endpoint Event Listener interface is *idempotent*. Endpoint Description Providers must inform an Endpoint Event Listener service (and its deprecated predecessor Endpoint Listener service) that is registered of all their matching Endpoints. The only way to find out about all available Endpoints is to register an Endpoint Event Listener (or Endpoint Listener) that is then informed by all available Endpoint Description Providers of their known Endpoint Descriptions that match their scope.

122.6.3 Endpoint Listener Interface

The `EndpointListener` interface is marked as `Deprecated` because the `EndpointEventListener` interface must be used in preference when both are implemented by the same object. The `EndpointEvent` interface has the following methods:

- `endpointAdded(EndpointDescription,String)` – Notify the Endpoint Listener of a new Endpoint Description. The second parameter is the filter that matched the Endpoint Description. Registering the same Endpoint multiple times counts as a single registration.
- `endpointRemoved(EndpointDescription,String)` – Notify the Endpoint Listener that the provided Endpoint Description is no longer available.

These methods must only be called if the Endpoint Listener service has a filter in its scope that matches the Endpoint Description properties. The reason for the filter string in the methods is to simplify and speed up matching an Endpoint Description to the cause of interest. For example, if the Listener Hook is used to do on demand import of services, then the filter can be associated with the Listener Info of the hook, see *On Demand* on page 511. If multiple filters in the scope match the Endpoint Description than the first filter in the scope must be passed.

The Endpoint Listener interface is *idempotent*. Endpoint Description Providers must inform an Endpoint Listener service that is registered of all their matching Endpoints.

122.6.4 Endpoint Event Listener and Endpoint Listener Implementations

An Endpoint Event Listener service tracks the known Endpoints in its given scope. There are potentially a large number of bundles involved in creating this federated registry of Endpoints. To ensure that no Endpoint Descriptions are orphaned or unnecessarily missed, an Endpoint Event Listener implementation must follow the following rules:

- *Registration* – The Endpoint Event Listener service is called with an event of type ADDED for all known Endpoint Descriptions that the bundles in the local framework are aware of. Similarly, Endpoint Listener services are called with an endpointAdded(EndpointDescription,String) method for all these.
- *Tracking providers* – An Endpoint Event Listener or Endpoint Listener must track the bundles that provide it with Endpoint Descriptions. If a bundle that provided Endpoint Descriptions is stopped, all Endpoint Descriptions that were provided by that bundle must be removed. This can be implemented straightforwardly with a Service Factory.
- *Scope modification* – An Endpoint Event Listener or Endpoint Listener is allowed to modify the set of filters in its scope through a service property modification. This modification must result in new and/or existing Endpoint Descriptions to be added, however, existing Endpoints that are no longer in scope are not required to be explicitly removed by the their sources. It is the responsibility for the Endpoint Listener to remove these orphaned Endpoint Description from its view.
- *Endpoint mutability* – An Endpoint Description can change its Properties. The way this is handled is different for Endpoint Event Listeners and Endpoint Listeners. An Endpoint Event Listener receives a change event of type MODIFIED when the Properties of an existing Endpoint are modified. If the modification means that the Endpoint no longer matches the listener scope an event of type MODIFIED_ENDMATCH is sent instead. Endpoint Listener services receive a sequence of endpointRemoved(EndpointDescription,String) and endpointAdded(EndpointDescription,String) callbacks when the Properties of an Endpoint are modified.

Endpoint Descriptions can be added from different sources and providers of Endpoint Descriptions often use asynchronous and potentially unreliable communications. An implementation must therefore handle the addition of multiple equal Endpoint Descriptions from different sources as well as from the same source. Implementations must not count the number of registrations, a remove operation of an Endpoint Description is final for each source. That is, if source A added Endpoint Description e, then it can only be removed by source A. However, if source A added e multiple times, then it only needs to be removed once. Removals of Endpoint Descriptions that have not been added (or were removed before) should be ignored.

The discovery of Endpoints is a fundamentally indeterministic process and implementations of Endpoint Event Listener services should realize that there are no guarantees that an added Endpoint Description is always describing a valid Endpoint.

122.6.5 Endpoint Description Providers

The Endpoint Event Listener and Endpoint Listener services are based on an asynchronous, unreliable, best effort model because there are few guarantees in a distributed world. It is the task of an Endpoint Description Provider, for example a discovery bundle, to keep the Endpoint Event Listener services up to date of any Endpoint Descriptions the provider is aware of and that match the tracked service's scope.

If an Endpoint Event Listener or Endpoint Listener service is registered, a provider must add all matching Endpoint Descriptions that it is aware of and match the tracked listener's scope. This can be done during registration or asynchronously later. For example, it is possible to use the filters in the scope to request remote systems for any Endpoint Descriptions that match those filters. For expediency reasons, the service registration event should not be delayed until those results return; it is therefore applicable to add these Endpoint Descriptions later when the returns from the remote systems finally arrive.

If a tracked listener service object is advertised as both an Endpoint Event Listener and an Endpoint Listener then the EndpointDescription Provider must ignore the EndpointListener interface, and treat the listener only as an Endpoint Event Listener. Remote Service Admin 1.0 actors will be unaware of the EndpointEventListener interface, and will treat the service object purely as an Endpoint Listener. This restriction ensures that all actors will treat the service either as an Endpoint Event Listener, or an Endpoint Listener, but never as both. As a result the listener service will not have to disambiguate duplicate events from a single source. If an Endpoint Description Provider uses both the Endpoint Listener and Endpoint Event Listener interfaces of a single service object then the resulting behavior is undefined. The implementation may detect the misuse and throw an Exception, process or ignore the events from one of the interfaces, or it may simply corrupt the internal registry of Endpoints within the listener.

A tracked Endpoint Event Listener or Endpoint Listener is allowed to modify its scope by setting new properties on its Service Registration. An Endpoint Description provider must process the new scope and add any newly matching Endpoint Descriptions. It is not necessary to remove any Endpoint Descriptions that were added before but no longer match the new scope. Removing those orphaned descriptions is the responsibility of the listener implementation.

It is not necessary to remove any registered Endpoint Descriptions when the Endpoint Event Listener or Endpoint Listener is unregistered; also here it is the responsibility of the listener to do the proper cleanup.

122.6.6 On Demand

A common distribution policy is to import services that are being listened for by local bundles. For example, when a bundle opens a Service Tracker on the Log Service, a Topology Manager could be notified and attempt to find a Log Service in the local cluster and then import this service in the local Service Registry.

The OSGi framework provides service hooks for exactly this purpose. A Topology Manager can register a Listener Hook service and receive the information about bundles that have specified an interests in specific services.

For example, a bundle creates the following Service Tracker:

```
ServiceTracker st = new ServiceTracker(context,
        LogService.class.getName() );
st.open();
```

This Service Tracker will register a Service Listener with the OSGi framework. This will cause the framework to add a ListenerInfo to any Listener Hook services. The getFilter method on a ListenerInfo object provides a filter that is directly applicable for the Endpoint Event Listener's scope. In the previous example, this would be the filter:

```
(objectClass=org.osgi.service.log.LogService)
```

A Topology Manager could verify if this listener is satisfied. That is, if it has at least one service. If no such service could be found, it could then add this filter to its Endpoint Event Listener's scope to detect remote implementations of this service. If such an Endpoint is detected, it could then request the import of this service through the Remote Service Admin service.

122.7 Events

The Remote Service Admin service must synchronously inform any Remote Service Admin Listener services of events as they happen. Client of the events should return quickly and not perform any but trivial processing in the same thread.

The following event types are defined:

- EXPORT_ERROR - An exported service has run into an unrecoverable error, although the Export Registration has not been closed yet. The event carries the Export Registration as well as the Exception that caused the problem, if present.
- EXPORT_REGISTRATION - The Remote Service Admin has registered a new Export Registration.
- EXPORT_UNREGISTRATION - An Export Registration has been closed, the service is no longer exported and the Endpoint is no longer active when this was the last registration for that service/Endpoint combination.
- EXPORT_UPDATE - An exported service is updated. The service properties have changed.
- EXPORT_WARNING - An exported service is experiencing problems but the Endpoint is still available.
- IMPORT_ERROR - An imported service has run into a fatal error and has been shut down. The Import Registration should be closed by the Topology Manager that created them.
- IMPORT_REGISTRATION - A new Import Registration was created for a potentially existing service/Endpoint combination.
- IMPORT_UNREGISTRATION - An Import Registration was closed, removing the proxy if this was the last registration.
- IMPORT_UPDATE - An imported service is updated. The service properties have changed.
- IMPORT_WARNING - An imported service is experiencing problems but can continue to function.

The following properties are available on the event:

- getType() - The type of the event.
- getException() - Any exception, if present.
- getExportReference() - An export reference, if applicable.
- getImportReference() - An import reference, if applicable.
- getSource() - The source of the event, the Remote Service Admin service.

122.7.1 Event Admin Mapping

All Remote Service Admin events must be posted, which is asynchronously, to the Event Admin service, if present, under the following topic:

```
org/osgi/service/remoteserviceadmin/<type>
```

Where <type> represents the type of the event, for example IMPORT_ERROR.

The Event Admin event must have the following properties:

- bundle - (Bundle) The Remote Service Admin bundle
- bundle.id - (Long) The id of the Remote Service Admin bundle.
- bundle.symbolicname - (String) The Bundle Symbolic Name of the Remote Service Admin bundle.version - (Version) The version of the Remote Service Admin bundle.
- bundle.signer - (String[]) Signer of the Remote Service Admin bundle
- exception - (Throwable) The Exception, if present. Also reported on the cause property for backward compatibility.
- exception.class - (String) The fully-qualified class name of the attached Exception.
- exception.message -(String) The message of the attached exception. Only set if the Exception message is not null.
- endpoint.service.id - (Long) Remote service id, if present
- endpoint.framework.uuid - (String) Remote service's Framework UUID, if present
- endpoint.id - (String) The id of the Endpoint, if present
- objectClass - (String[]) The interface names, if present

- service.imported.configs - (String+) The configuration types of the imported services, if present
- timestamp - (Long) The time when the event occurred
- event - (RemoteServiceAdminEvent) The RemoteServiceAdminEvent object that caused this event.

122.8 Endpoint Description Extender Format

The Endpoint Description Extender format is a possibility to deliver Endpoint Descriptions in bundles. This section defines an XML schema and how to locate XML definition resources that use this schema to define Endpoint Descriptions. The definition resource is a simple property based model that can define the same information as the properties on an imported service. If a bundle with the description is *ready* (ACTIVE or lazy activation and in the STARTING state), then this static description can be disseminated through the Endpoint Event Listeners that have specified an interest in this description. If the bundle is stopped, the corresponding Endpoints must be removed.

XML documents containing remote service descriptions must be specified by the Remote-Service header in the manifest. The structure of the Remote Service header is:

```
Remote-Service ::= header // See Common Header Syntax in Core
```

The value of the header is a comma separated list of paths. A path is:

- A directory if it ends with a solidus ('/' \u002F). A directory is scanned for *.xml files.
- A path with wildcards. Such a path can use the wildcards in its last component, as defined in the findEntries method.
- A complete path, not having wildcards not ending in a solidus ('/' \u002F).

The Remote-Service header has no architected directives or attributes, unrecognized attributes and directives must be ignored.

A Remote-Service manifest header specified in a fragment must be ignored. However, XML documents referenced by a bundle's Remote-Service manifest header can be contained in attached fragments. The required behavior for this is implemented in the findEntries method.

The extender must process each XML document specified in this header. If an XML document specified by the header cannot be located in the bundle and its attached fragments, the extender must log an error message with the Log Service, if present, and continue.

For example:

```
Remote-Service: lib/, remote/osgi/*.dsc, cnf/google.xml
```

This matches all resources in the lib directory matching *.xml, all resources in the /remote/osgi directory that end with .dsc, as well as the google.xml resource in the cnf directory.

The namespace of these XML resources must be:

```
 http://www.osgi.org/xmlns/rsa/v1.0.0
```

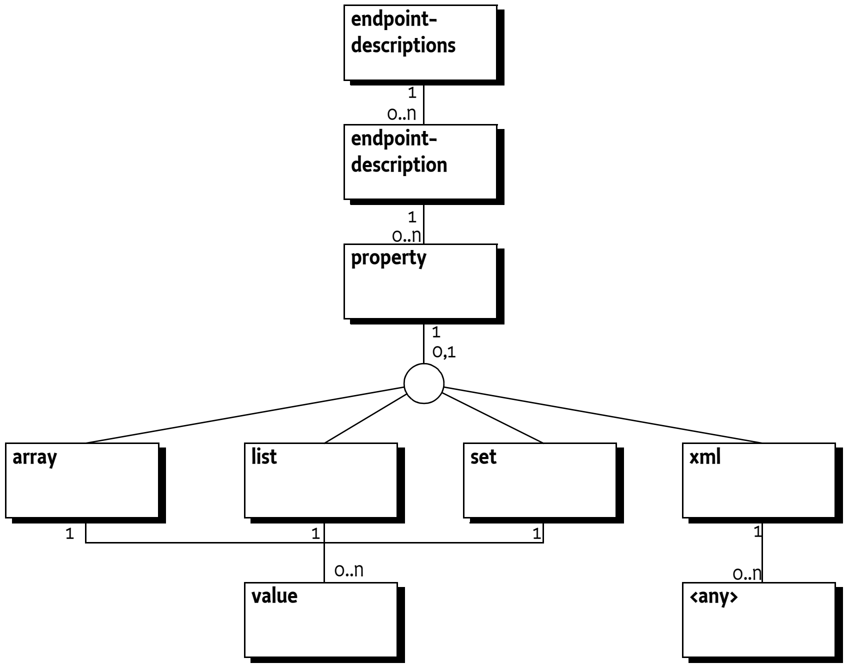
This namespace describes a set of Endpoint Descriptions, where each Endpoint Description can provide a set of properties. The structure of this schema is:

```
endpoint-descriptions  ::= <endpoint-description>*
endpoint-description   ::= <property>*
property               ::= ( <array> | <list> | <set>| <xml> )?
array                  ::= <value> *
```

```
list                    ::= <value> *
set                     ::= <value> *
xml                     ::= <*> *
```

This structure is depicted in Figure 122.6 on page 514.

Figure 122.6 *Endpoint Description XML Structure*



The property element has the attributes listed in table Table 122.2.

Table 122.2 *Property Attributes*

Attribute	Type	Description
name	String	The required name of the property. The type maps to the XML Schema xsd:string type.

Attribute	Type	Description
value-type	String | long | Long | double | Double | float | Float | int | Integer | byte | Byte | char | Character | boolean | Boolean | short | Short	The optional type name of the property, the default is String. Any value in the value attribute or the value element when collections are used must be converted to the corresponding Java types. If the primitive form, for example byte, is specified for non-array types, then the value must be silently converted to the corresponding wrapper type.
value	String	The value. Must be converted to the specified type if this is not the String type. The value attribute must not be used when the property element has a child element.

A property can have an array, list, set, or xml child element. If a child element is present then it is an error if the value attribute is defined. It is also an error of there is no child element and no value attribute.

The array, list, or set are *multi-valued*. That is, they contain 0 or more value elements. A value element contains text (a string) that must be converted to the given value-type or if not specified, left as is. Conversion must *trim* the leading and trailing white space characters as defined in the Character.isWhitespace method. No trimming must be done for strings. An array of primitive integers like int[] {1,42,97} can be encoded as follows:

```
<property name="integers" value-type="int">
    <array>
        <value> 1</value>
        <value>42</value>
        <value>97</value>
    </array>
</property>
```

The xml element is used to convey XML from other namespaces, it is allowed to contain one foreign XML root element, with any number of children, that will act as the root element of an XML document. This root element will be included in the corresponding property as a string. The XML element must be a valid XML document but not contain the XML processing instructions, the part between the <? and ?>. The value-type of the property must be String or not set when an xml element is used, using another type is invalid.

The xml element can be used to embed configuration information, making the Endpoint Description self contained.

The following is an example of an endpoint-descriptions resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">
    <endpoint-description>
        <property name="service.intents">
            <list>
                <value>SOAP</value>
                <value>HTTP</value>
            </list>
        </property>
        <property name="endpoint.id" value="http://ws.acme.com:9000/hello"/>
        <property name="endpoint.package.version.com.acme" value="4.2"/>
        <property name="objectClass">
            <array>
                <value>com.acme.Foo</value>
            </array>
        </property>
        <property name="service.imported.configs" value="com.acme"/>
        <property name="com.acme.ws.xml">
            <xml>
                <config xmlns="http://acme.com/defs">
                    <port>1029</port>
                    <host>www.acme.com</host>
                </config>
            </xml>
        </property>
    </endpoint-description>
</endpoint-descriptions>
```

Besides being in a separate resource, the static configuration as described here could also be part of a larger XML file. In that case the parser must ignore elements not part of the http://www.osgi.org/xmlns/rsa/v1.0.0 namespace schema.

122.8.1 XML Schema

This namespace of the schema is:

```
http://www.osgi.org/xmlns/rsa/v1.0.0
```

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:rsa="http://www.osgi.org/xmlns/rsa/v1.0.0"
  targetNamespace="http://www.osgi.org/xmlns/rsa/v1.0.0"
  elementFormDefault="qualified" version="1.0.1">

  <annotation>
    <documentation xml:lang="en">
      This is the XML Schema for endpoint descriptions used by
      the Remote Service Admin Specification. Endpoint descriptions
      are used to describe remote services to a client in cases
      where a real live Discovery system isn't used. An extender,
      such as a local Discovery Service can look for service
      descriptions in installed bundles and inform the Topology
      Manager of these remote services. The Topology Manager can then
      instruct the Remote Service Admin to create client proxies for
      these services.
    </documentation>
  </annotation>

  <element name="endpoint-descriptions" type="rsa:Tendpoint-descriptions" />

  <complexType name="Tendpoint-descriptions">
    <sequence>
      <element name="endpoint-description" type="rsa:Tendpoint-description"
               minOccurs="1" maxOccurs="unbounded" />
        <!--
          It is non-deterministic, per W3C XML Schema 1.0:
          http://www.w3.org/TR/xmlschema-1/#cos-nonambig to use
          namespace="##any" below.
        -->
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tendpoint-description">
    <annotation>
      <documentation xml:lang="en">
        A Distribution Provider can register a proxy with the properties
        provided. Whether or not it is instructed to do so, is up to the
        Topology Manager. If any 'intents' properties are specified then the
        Distribution Provider should only register a proxy if it can support
        those intents.
      </documentation>
    </annotation>
    <sequence>
      <element name="property" type="rsa:Tproperty" minOccurs="1"
               maxOccurs="unbounded" />
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tproperty" mixed="true">
    <sequence>
      <choice minOccurs="0" maxOccurs="1">
        <element name="array" type="rsa:Tmulti-value"/>
        <element name="list" type="rsa:Tmulti-value"/>
        <element name="set" type="rsa:Tmulti-value"/>
        <element name="xml" type="rsa:Txml"/>
      </choice>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="optional" />
    <attribute name="value-type" type="rsa:Tvalue-types" default="String" use="optional" />
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tmulti-value">
    <sequence>
      <element name="value" minOccurs="0" maxOccurs="unbounded" type="rsa:Tvalue"/>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tvalue" mixed="true">
    <sequence>
      <element name="xml" minOccurs="0" maxOccurs="1" type="rsa:Txml"/>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <!-- Specifies the data type of a property or of the elements in a multi-value
       property. Numerical and boolean values are trimmed before they are processed.
       Simple types are automatically boxed if needed. Only the array data type
       allows for simple type values. When specifying a simple type on any other
       type of property it will automatically be boxed. -->
  <simpleType name="Tvalue-types">
    <restriction base="string">
      <enumeration value="String" />
```

```
      <enumeration value="long" />
      <enumeration value="Long" />
      <enumeration value="double" />
      <enumeration value="Double" />
      <enumeration value="float" />
      <enumeration value="Float" />
      <enumeration value="int" />
      <enumeration value="Integer" />
      <enumeration value="byte" />
      <enumeration value="Byte" />
      <enumeration value="char" />
      <enumeration value="Character" />
      <enumeration value="boolean" />
      <enumeration value="Boolean" />
      <enumeration value="short" />
      <enumeration value="Short" />
    </restriction>
  </simpleType>

  <!-- This complex type allows literal XML to be specified in an <xml/> tag (which
      is more convenient than putting it in a CDATA section).
      The embedded XML must be well-formed and not be in the rsa namespace. It will
      be put in a String value of a property or in an element of a multi-value
      property of base type String. The XML will be prefixed with the standard
      <?XML ?> header which is copied from the enclosing document. Hence it will
      carry the same version and encoding as the document in the rsa namespace. -->
  <complexType name="Txml">
    <sequence>
      <any namespace="##other" minOccurs="1" maxOccurs="1"
          processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <attribute name="must-understand" type="boolean" default="false">
    <annotation>
      <documentation xml:lang="en">
        This attribute should be used by extensions to documents
        to require that the document consumer understand the
        extension.
      </documentation>
    </annotation>
  </attribute>
</schema>
```

122.9 Capability Namespaces

122.9.1 Local Discovery Extender

A bundle containing Endpoint Description Extender resources can indicate its dependency on the Remote Service Admin extender by declaring a requirement on the osgi.extender namespace.

```
Require-Capability: osgi.extender;
    filter:="(&(osgi.extender=osgi.remoteserviceadmin.localdiscovery)
             (version>=1.0)(!(version>=2.0)))"
```

With this constraint declared a bundle that depends on the extender will fail to resolve if no extender is present in the framework.

Implementations of this specification must provide this extender capability at version 1.0 as follows:

```
Provide-Capability: osgi.extender;
    osgi.extender="osgi.remoteserviceadmin.localdiscovery";
    version:Version="1.0";
    uses:="org.osgi.service.remoteserviceadmin"
```

The reason that the extender capability is declared at version 1.0 is because the extender is unchanged from version 1.0 of this specification.

122.9.2 Discovery Provider Capability

Discovery Providers use the osgi.remoteserviceadmin.discovery namespace to declare themselves as such. The version defined for this namespace indicates the version of this specification that the discovery provider supports.

This namespace has a defined attribute, protocols of type List<String>, which contains a list of the discovery protocols supported by the discovery provider. Local discovery providers (using the *Endpoint Description Extender Format* on page 513), should use the value local to indicate that they support this. Additionally, it defines a version attribute. Other values for the protocols attribute are implementation specific.

Table 122.3 osgi.remoteserviceadmin.discovery Namespace

Name	Kind	M/O	Type	Syntax	Description
protocols	CA	M	List<String>	symbolic-name	The discovery protocols supported. A value of local indicates support for the *Endpoint Description Extender Format* on page 513.
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A discovery provider that provides local and SLP discovery:

```
Provide-Capability: osgi.remoteserviceadmin.discovery;
    protocols:List<String>="SLP,local"; version:Version=1.1
```

122.9.3 Distribution Provider Capability

Distribution providers advertise their supported distribution mechanisms using configuration types. These are selected at runtime using the service.exported.configs service property. Distribution providers can use the osgi.remoteserviceadmin.distribution namespace with attribute configs, of type List<String>, to advertise the supported config types.

Table 122.4 osgi.remoteserviceadmin.distribution Namespace

Name	Kind	M/O	Type	Syntax	Description
configs	CA	M	List<String>	symbolic-name	Supported configuration types. See *Endpoint Description* on page 498 .
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A Distribution provider that supports the org.acme.jaxws and org.acme.jaxrs configuration types:

```
Provide-Capability: osgi.remoteserviceadmin.distribution;
    configs:List<String>="org.acme.jaxws,org.acme.jaxrs"; version:Version=1.1
```

122.9.4 Topology Manager Capability

Remote Service Admin topology managers may use different policies when determining which services to export and/or import. Topology managers use the namespace osgi.remoteserviceadmin.topology to declare this behavior. This namespace defines the policy attribute of type List<String>. Values are implementation specific, but example definitions can be found at *Example Use Cases* on page 497.

Table 122.5 *osgi.remoteserviceadmin.topology Namespace*

Name	Kind	M/O	Type	Syntax	Description
policy	CA	M	List<String>	symbolic-name	The policy used for importing and exporting services. In general the policy is implementation specific.
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A Topology manager that supports a *promiscuous* policy:

```
Provide-Capability: osgi.remoteserviceadmin.topology;
    policy:List<String>=promiscuous; version:Version=1.1
```

122.9.5 Service Capability

The Distribution Provider provides the *Remote Service Admin* service. To inform tools about this service it must provide the osgi.service namespace representing the RemoteServiceAdmin service. This capability must also declare a uses constraint for the org.osgi.service.remoteserviceadmin package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>=
    "org.osgi.service.remoteserviceadmin.RemoteServiceAdmin";
  uses:="org.osgi.service.remoteserviceadmin"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

122.10 Advice to implementations

This section is not intended to be normative, but offers advice to implementations as to how the complexity of supporting both the new Endpoint Event Listener and Endpoint Listener services can be managed and minimized. This advice applies to both Discovery Providers and Topology Managers implementing Remote Service Admin 1.1.

122.10.1 Notifying listeners

Endpoint Event Listeners and Endpoint Listeners have a very similar behavior and lifecycle. They also use the same property names to define their scope filter. It is therefore relatively simple for an Endpoint Description Provider to notify both Endpoint Listener and Endpoint Event Listeners using a single code path.

One possible mechanism is to track both the listener types using the same Service Tracker. If the tracked Service Reference advertises the EndpointEventListener interface then it must be treated as an Endpoint Event Listener. If not then the Endpoint Listener service can be wrapped in an adapter that converts Endpoint Event Listener events into the appropriate Endpoint Listener calls. The main notification code path can then treat every listener as an Endpoint Event Listener.

122.10.2 Receiving Endpoint lifecycle notifications

The Remote Service Admin 1.1 specification is backward compatible with version 1.0, meaning that version 1.1 actors must register an Endpoint Listener service. There is no restriction requiring this listener to be the same service as the Endpoint Event Listener, however there is a significant advantage to combining the listeners into a single service registration.

By making the two listeners a single service object a bundle can guarantee that it will not receive multiple notifications for the same event. If the service registrations are separate then Endpoint Description Providers will see two separate listeners, and notify them both. As a single service registration only one event will occur, and using the highest mutually supported version of the Remote Service Admin Specification.

122.11 Security

From a security point of view distribution is a significant threat. A Distribution Provider requires very significant capabilities to be able to proxy services. In many situations it will be required to grant the distribution provider All Permission. It is therefore highly recommended that Distribution Providers use trusted links and ensure that it is not possible to attack a system through the Remote Services Admin service and used discovery protocols.

122.11.1 Import and Export Registrations

Import and Export Registrations are *capabilities*. That is, they can only be obtained when the caller has the proper permissions but once obtained they are no longer checked. The caller should therefore be careful to share those objects with other bundles. Export and Import References are free to share.

122.11.2 Endpoint Permission

The Remote Service Admin implementation requires a large set of permissions because it must be able to distribute potentially any service. Giving these extensive capabilities to all Topology Managers would make it harder to developer general Topology Managers that implements specific scenarios. For this reason, this specification provides an Endpoint Permission.

When an Endpoint Permission must be verified, it must be created with an Endpoint Description as argument, like:

```
sm.checkPermission( new EndpointPermission(anEndpoint,localUUID,READ));
```

The standard name and action constructor is used to define a permission. The name argument is a filter expression. The filter for an Endpoint Permission is applied to the properties of an Endpoint Description. The localUUID must map to the UUID of the framework of the caller of this constructor, see *Framework UUID* on page 501. This localUUID is used to allow a the permissions to use the <<LOCAL>> magic name in the permission filter to refer to the local framework.

The filter expression can use the following magic value:

- <<LOCAL>> - This value represents the framework UUID of the framework that this bundle belongs to. The following example restricts the visibility to descriptions of local Endpoints:

```
ALLOW {
  ...EndpointPermission
       "(endpoint.framework.uuid=<<LOCAL>>)"
       "READ" }
```

An Endpoint Permission that has the actions listed in the following table.

Table 122.6 Endpoint Permission Actions

Action	Methods	Description
IMPORT	importService(EndpointDescription)	Import an Endpoint
EXPORT	exportService(ServiceReference,Map)	Export a service

Action	Methods	Description

READ	getExportedServices()
	getImportedEndpoints()
	remoteAdminEvent(RemoteServiceAdminEvent)
		See the presence of distributed services. The IMPORT and EXPORT action imply READ. Distribution of events to the Remote Service Admin Listener. The Remote Service Admin must verify that the listener's bundle has the proper permission. No events should be delivered that are not implied.

122.12 org.osgi.service.remoteserviceadmin

Remote Service Admin Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.remoteserviceadmin; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.remoteserviceadmin; version="[1.1,1.2)"
```

122.12.1 Summary

- EndpointDescription - A description of an endpoint that provides sufficient information for a compatible distribution provider to create a connection to this endpoint An Endpoint Description is easy to transfer between different systems because it is property based where the property keys are strings and the values are simple types.
- EndpointEvent - An Endpoint Event.
- EndpointEventListener - A white board service that represents a listener for endpoints.
- EndpointListener - Deprecated white board service that represents a listener for endpoints.
- EndpointPermission - A bundle's authority to export, import or read an Endpoint.
- ExportReference - An Export Reference associates a service with a local endpoint.
- ExportRegistration - An Export Registration associates a service to a local endpoint.
- ImportReference - An Import Reference associates an active proxy service to a remote endpoint.
- ImportRegistration - An Import Registration associates an active proxy service to a remote endpoint.
- RemoteConstants - Provide the definition of the constants used in the Remote Service Admin specification.
- RemoteServiceAdmin - A Remote Service Admin manages the import and export of services.
- RemoteServiceAdminEvent - Provides the event information for a Remote Service Admin event.
- RemoteServiceAdminListener - A RemoteServiceAdminEvent listener is notified synchronously of any export or import registrations and unregistrations.

122.12.2 public class EndpointDescription

A description of an endpoint that provides sufficient information for a compatible distribution provider to create a connection to this endpoint An Endpoint Description is easy to transfer between different systems because it is property based where the property keys are strings and the values are simple types. This allows it to be used as a communications device to convey available endpoint information to nodes in a network. An Endpoint Description reflects the perspective of an *importer*. That is, the property keys have been chosen to match filters that are created by client bundles that need a service. Therefore the map must not contain any `service.exported.*` property and must contain the corresponding `service.imported.*` ones. The `service.intents` property must contain the intents provided by the service itself combined with the intents added by the exporting distribution provider. Qualified intents appear fully expanded on this property.

Concurrency Immutable

122.12.2.1 public EndpointDescription(Map<String, ?> properties)

properties The map from which to create the Endpoint Description. The keys in the map must be type `String` and, since the keys are case insensitive, there must be no duplicates with case variation.

□ Create an Endpoint Description from a Map.

The endpoint.id, service.imported.configs and `objectClass` properties must be set.

Throws `IllegalArgumentException`– When the properties are not proper for an Endpoint Description.

122.12.2.2 public EndpointDescription(ServiceReference<?> reference, Map<String, ?> properties)

reference A service reference that can be exported.

properties Map of properties. This argument can be `null`. The keys in the map must be type `String` and, since the keys are case insensitive, there must be no duplicates with case variation.

□ Create an Endpoint Description based on a Service Reference and a Map of properties. The properties in the map take precedence over the properties in the Service Reference.

This method will automatically set the endpoint.framework.uuid and endpoint.service.id properties based on the specified Service Reference as well as the service.imported property if they are not specified as properties.

The endpoint.id, service.imported.configs and `objectClass` properties must be set.

Throws `IllegalArgumentException`– When the properties are not proper for an Endpoint Description

122.12.2.3 public boolean equals(Object other)

other The `EndpointDescription` object to be compared.

□ Compares this `EndpointDescription` object to another object.

An Endpoint Description is considered to be **equal to** another Endpoint Description if their ids are equal.

Returns `true` if `object` is a `EndpointDescription` and is equal to this object; `false` otherwise.

122.12.2.4 public List<String> getConfigurationTypes()

□ Returns the configuration types. A distribution provider exports a service with an endpoint. This endpoint uses some kind of communications protocol with a set of configuration parameters. There are many different types but each endpoint is configured by only one configuration type. However, a distribution provider can be aware of different configuration types and provide synonyms to increase the change a receiving distribution provider can create a connection to this endpoint. This value of the configuration types is stored in the RemoteConstants.SERVICE_IMPORTED_CONFIGS service property.

Returns An unmodifiable list of the configuration types used for the associated endpoint and optionally synonyms.

122.12.2.5 public String getFrameworkUUID()

□ Return the framework UUID for the remote service, if present. The value of the remote framework UUID is stored in the RemoteConstants.ENDPOINT_FRAMEWORK_UUID endpoint property.

Returns Remote Framework UUID, or `null` if this endpoint is not associated with an OSGi framework having a framework UUID.

122.12.2.6 public String getId()

□ Returns the endpoint's id. The id is an opaque id for an endpoint. No two different endpoints must have the same id. Two Endpoint Descriptions with the same id must represent the same endpoint. The value of the id is stored in the RemoteConstants.ENDPOINT_ID property.

Returns The id of the endpoint, never `null`. The returned value has leading and trailing whitespace removed.

122.12.2.7 public List<String> getIntents()

□ Return the list of intents implemented by this endpoint. The intents are based on the service.intents on an imported service, except for any intents that are additionally provided by the importing distribution provider. All qualified intents must have been expanded. This value of the intents is stored in the RemoteConstants.SERVICE_INTENTS service property.

Returns An unmodifiable list of expanded intents that are provided by this endpoint.

122.12.2.8 public List<String> getInterfaces()

□ Provide the list of interfaces implemented by the exported service. The value of the interfaces is derived from the `objectClass` property.

Returns An unmodifiable list of Java interface names implemented by this endpoint.

122.12.2.9 public Version getPackageVersion(String packageName)

packageName The name of the package for which a version is requested.

□ Provide the version of the given package name. The version is encoded by prefixing the given package name with endpoint.package.version., and then using this as an endpoint property key. For example:

```
endpoint.package.version.com.acme
```

The value of this property is in String format and will be converted to a `Version` object by this method.

Returns The version of the specified package or `Version.emptyVersion` if the package has no version in this Endpoint Description.

Throws `IllegalArgumentException`– If the version property value is not String.

122.12.2.10 public Map<String, Object> getProperties()

□ Returns all endpoint properties.

Returns An unmodifiable map referring to the properties of this Endpoint Description.

122.12.2.11 public long getServiceId()

□ Returns the service id for the service exported through this endpoint. This is the service id under which the framework has registered the service. This field together with the Framework UUID is a globally unique id for a service. The value of the remote service id is stored in the RemoteConstants.ENDPOINT_SERVICE_ID endpoint property.

Returns Service id of a service or 0 if this Endpoint Description does not relate to an OSGi service.

122.12.2.12 public int hashCode()

□ Returns a hash code value for the object.

Returns An integer which is a hash code value for this object.

122.12.2.13 public boolean isSameService(EndpointDescription other)

other The Endpoint Description to look at

□ Answers if this Endpoint Description refers to the same service instance as the given Endpoint Description. Two Endpoint Descriptions point to the same service if they have the same id or their framework UUIDs and remote service ids are equal.

Returns True if this endpoint description points to the same service as the other

122.12.2.14 public boolean matches(String filter)

filter The filter to test.

□ Tests the properties of this EndpointDescription against the given filter using a case insensitive match.

Returns true If the properties of this EndpointDescription match the filter, false otherwise.

Throws IllegalArgumentException– If filter contains an invalid filter string that cannot be parsed.

122.12.2.15 public String toString()

□ Returns the string representation of this EndpointDescription.

Returns String form of this EndpointDescription.

122.12.3 public class EndpointEvent

An Endpoint Event.

EndpointEvent objects are delivered to all registered EndpointEventListener services where the EndpointDescription properties match one of the filters specified in the EndpointEventListener.ENDPOINT_LISTENER_SCOPE registration properties of the Endpoint Event Listener.

A type code is used to identify the type of event. The following event types are defined:

- ADDED
- REMOVED
- MODIFIED
- MODIFIED_ENDMATCH

Additional event types may be defined in the future.

See Also EndpointEventListener

Since 1.1

Concurrency Immutable

122.12.3.1 public static final int ADDED = 1

An endpoint has been added.

This EndpointEvent type indicates that a new endpoint has been added. The endpoint is represented by the associated EndpointDescription object.

122.12.3.2 public static final int MODIFIED = 4

The properties of an endpoint have been modified.

This EndpointEvent type indicates that the properties of an existing endpoint have been modified. The endpoint is represented by the associated EndpointDescription object and its properties can be obtained via EndpointDescription.getProperties(). The endpoint properties still match the filters as specified in the EndpointEventListener.ENDPOINT_LISTENER_SCOPE filter.

122.12.3.3 public static final int MODIFIED_ENDMATCH = 8

The properties of an endpoint have been modified and the new properties no longer match the listener's filter.

This EndpointEvent type indicates that the properties of an existing endpoint have been modified and no longer match the filter. The endpoint is represented by the associated EndpointDescription object and its properties can be obtained via EndpointDescription.getProperties(). As a consequence of the modification the filters as specified in the EndpointEventListener.ENDPOINT_LISTENER_SCOPE do not match any more.

122.12.3.4 public static final int REMOVED = 2

An endpoint has been removed.

This EndpointEvent type indicates that an endpoint has been removed. The endpoint is represented by the associated EndpointDescription object.

122.12.3.5 public EndpointEvent(int type, EndpointDescription endpoint)

type The event type. See getType().

endpoint The endpoint associated with the event.

□ Constructs a EndpointEvent object from the given arguments.

122.12.3.6 public EndpointDescription getEndpoint()

□ Return the endpoint associated with this event.

Returns The endpoint associated with the event.

122.12.3.7 public int getType()

□ Return the type of this event.

The type values are:

- ADDED
- REMOVED
- MODIFIED
- MODIFIED_ENDMATCH

Returns The type of this event.

122.12.4 public interface EndpointEventListener

A white board service that represents a listener for endpoints. An Endpoint Event Listener represents a participant in the distributed model that is interested in Endpoint Descriptions. This white board service can be used in many different scenarios. However, the primary use case is to allow a remote manager to be informed of Endpoint Descriptions available in the network and inform the network about available Endpoint Descriptions. Both the network bundle and the manager bundle register an Endpoint Event Listener service. The manager informs the network bundle about Endpoints that it creates. The network bundles then uses a protocol like SLP to announce these local end-points to the network. If the network bundle discovers a new Endpoint through its discovery protocol, then it sends an Endpoint Description to all the Endpoint Listener services that are registered (except its own) that have specified an interest in that endpoint. Endpoint Event Listener services can express their *scope* with the service property ENDPOINT_LISTENER_SCOPE. This service property is a list of filters. An Endpoint Description should only be given to a Endpoint Event Listener when there is at least one filter that matches the Endpoint Description properties. This filter model is quite flexible. For example, a discovery bundle is only interested in locally originating Endpoint Descriptions. The following filter ensures that it only sees local endpoints.

```
(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
```

In the same vein, a manager that is only interested in remote Endpoint Descriptions can use a filter like:

```
(!(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72))
```

Where in both cases, the given UUID is the UUID of the local framework that can be found in the Framework properties. The Endpoint Event Listener's scope maps very well to the service hooks. A manager can just register all filters found from the Listener Hook as its scope. This will automatically provide it with all known endpoints that match the given scope, without having to inspect the filter string. In general, when an Endpoint Description is discovered, it should be dispatched to all registered Endpoint Event Listener services. If a new Endpoint Event Listener is registered, it should be informed about all currently known Endpoints that match its scope. If a getter of the Endpoint Listener service is unregistered, then all its registered Endpoint Description objects must be removed. The Endpoint Event Listener models a *best effort* approach. Participating bundles should do their utmost to keep the listeners up to date, but implementers should realize that many endpoints come through unreliable discovery processes. The Endpoint Event Listener supersedes the EndpointListener interface as it also supports notifications around modifications of endpoints.

Since 1.1

Concurrency Thread-safe

122.12.4.1 public static final String ENDPOINT_LISTENER_SCOPE = "endpoint.listener.scope"

Specifies the interest of this listener with filters. This listener is only interested in Endpoint Descriptions where its properties match the given filter. The type of this property must be `String+`.

122.12.4.2 public void endpointChanged(EndpointEvent event, String filter)

event The event containing the details about the change.

filter The filter from the ENDPOINT_LISTENER_SCOPE that matches (or for EndpointEvent.MODIFIED_ENDMATCH and EndpointEvent.REMOVED used to match) the endpoint, must not be `null`.

□ Notification that an endpoint has changed. Details of the change is captured in the Endpoint Event provided. This could be that an endpoint was added, removed or modified.

122.12.5 public interface EndpointListener

Deprecated white board service that represents a listener for endpoints. An Endpoint Listener represents a participant in the distributed model that is interested in Endpoint Descriptions. The Endpoint Listener is called back when matching endpoints are added or removed. Consumers interested in the modification of endpoints, when associated service properties are changed, should use an EndpointEventListener instead. This white board service can be used in many different scenarios. However, the primary use case is to allow a remote manager to be informed of Endpoint Descriptions available in the network and inform the network about available Endpoint Descriptions. Both the network bundle and the manager bundle register an Endpoint Listener service. The manager informs the network bundle about Endpoints that it creates. The network bundles then uses a protocol like SLP to announce these local end-points to the network. If the network bundle discovers a new Endpoint through its discovery protocol, then it sends an Endpoint Description to all the Endpoint Listener services that are registered (except its own) that have specified an interest in that endpoint. Endpoint Listener services can express their *scope* with the service property ENDPOINT_LISTENER_SCOPE. This service property is a list of filters. An Endpoint Description should only be given to a Endpoint Listener when there is at least one filter that matches the Endpoint Description properties. This filter model is quite flexible. For example, a discovery bundle is only interested in locally originating Endpoint Descriptions. The following filter ensure that it only sees local endpoints.

```
(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
```

In the same vein, a manager that is only interested in remote Endpoint Descriptions can use a filter like:

```
(!(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72))
```

Where in both cases, the given UUID is the UUID of the local framework that can be found in the Framework properties. The Endpoint Listener's scope maps very well to the service hooks. A manager can just register all filters found from the Listener Hook as its scope. This will automatically provide it with all known endpoints that match the given scope, without having to inspect the filter string. In general, when an Endpoint Description is discovered, it should be dispatched to all registered Endpoint Listener services. If a new Endpoint Listener is registered, it should be informed about all currently known Endpoints that match its scope. If a getter of the Endpoint Listener service is unregistered, then all its registered Endpoint Description objects must be removed. The Endpoint Listener models a *best effort* approach. Participating bundles should do their utmost to keep the listeners up to date, but implementers should realize that many endpoints come through unreliable discovery processes.

Deprecated As of 1.1. Replaced by EndpointEventListener.

Concurrency Thread-safe

122.12.5.1 public static final String ENDPOINT_LISTENER_SCOPE = "endpoint.listener.scope"

Specifies the interest of this listener with filters. This listener is only interested in Endpoint Descriptions where its properties match the given filter. The type of this property must be `String+`.

122.12.5.2 public void endpointAdded(EndpointDescription endpoint, String matchedFilter)

endpoint The Endpoint Description to be published

matchedFilter The filter from the ENDPOINT_LISTENER_SCOPE that matched the endpoint, must not be `null`.

□ Register an endpoint with this listener. If the endpoint matches one of the filters registered with the ENDPOINT_LISTENER_SCOPE service property then this filter should be given as the `matchedFilter` parameter. When this service is first registered or it is modified, it should receive all known endpoints matching the filter.

122.12.5.3 public void endpointRemoved(EndpointDescription endpoint, String matchedFilter)

endpoint The Endpoint Description that is no longer valid.

matchedFilter The filter from the ENDPOINT_LISTENER_SCOPE that matched the endpoint, must not be `null`.

□ Remove the registration of an endpoint. If an endpoint that was registered with the endpointAdded(EndpointDescription, String) method is no longer available then this method should be called. This will remove the endpoint from the listener. It is not necessary to remove endpoints when the service is unregistered or modified in such a way that not all endpoints match the interest filter anymore.

122.12.6 public final class EndpointPermission extends Permission

A bundle's authority to export, import or read an Endpoint.

- The `export` action allows a bundle to export a service as an Endpoint.
- The `import` action allows a bundle to import a service from an Endpoint.

- The read action allows a bundle to read references to an Endpoint.

Permission to read an Endpoint is required in order to detect events regarding an Endpoint. Untrusted bundles should not be able to detect the presence of certain Endpoints unless they have the appropriate EndpointPermission to read the specific service.

Concurrency Thread-safe

122.12.6.1 public static final String EXPORT = "export"

The action string export. The export action implies the read action.

122.12.6.2 public static final String IMPORT = "import"

The action string import. The import action implies the read action.

122.12.6.3 public static final String READ = "read"

The action string read.

122.12.6.4 public EndpointPermission(String filterString, String actions)

filterString The filter string or "*" to match all endpoints.

actions The actions read, import, or export.

□ Create a new EndpointPermission with the specified filter.

The filter will be evaluated against the endpoint properties of a requested EndpointPermission.

There are three possible actions: read, import and export. The read action allows the owner of this permission to see the presence of distributed services. The import action allows the owner of this permission to import an endpoint. The export action allows the owner of this permission to export a service.

Throws IllegalArgumentException– If the filter has an invalid syntax or the actions are not valid.

122.12.6.5 public EndpointPermission(EndpointDescription endpoint, String localFrameworkUUID, String actions)

endpoint The requested endpoint.

localFrameworkUUID The UUID of the local framework. This is used to support matching the endpoint.framework.uuid endpoint property to the <<LOCAL>> value in the filter expression.

actions The actions read, import, or export.

□ Creates a new requested EndpointPermission object to be used by code that must perform checkPermission. EndpointPermission objects created with this constructor cannot be added to an EndpointPermission permission collection.

Throws IllegalArgumentException– If the endpoint is null or the actions are not valid.

122.12.6.6 public boolean equals(Object obj)

obj The object to test for equality.

□ Determines the equality of two EndpointPermission objects. Checks that specified object has the same name, actions and endpoint as this EndpointPermission.

Returns true If obj is a EndpointPermission, and has the same name, actions and endpoint as this EndpointPermission object; false otherwise.

122.12.6.7 public String getActions()

□ Returns the canonical string representation of the actions. Always returns present actions in the following canonical order: read, import, export.

Returns The canonical string representation of the actions.

122.12.6.8 public int hashCode()

□ Returns the hash code value for this object.

Returns Hash code value for this object.

122.12.6.9 public boolean implies(Permission p)

p The target permission to check.

□ Determines if a EndpointPermission object "implies" the specified permission.

Returns true if the specified permission is implied by this object; false otherwise.

122.12.6.10 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object for storing EndpointPermission objects.

Returns A new PermissionCollection object suitable for storing EndpointPermission objects.

122.12.7 public interface ExportReference

An Export Reference associates a service with a local endpoint. The Export Reference can be used to reference an exported service. When the service is no longer exported, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.7.1 public EndpointDescription getExportedEndpoint()

□ Return the Endpoint Description for the local endpoint.

Returns The Endpoint Description for the local endpoint. Must be null when the service is no longer exported.

122.12.7.2 public ServiceReference<?> getExportedService()

□ Return the service being exported.

Returns The service being exported. Must be null when the service is no longer exported.

122.12.8 public interface ExportRegistration

An Export Registration associates a service to a local endpoint. The Export Registration can be used to delete the endpoint associated with an this registration. It is created with the RemoteServiceAdmin.exportService(ServiceReference,Map) method. When this Export Registration has been closed, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.8.1 public void close()

□ Delete the local endpoint and disconnect any remote distribution providers. After this method returns, all methods must return null. This method has no effect when this registration has already been closed or is being closed.

122.12.8.2 public Throwable getException()

□ Return the exception for any error during the export process. If the Remote Service Admin for some reasons is unable to properly initialize this registration, then it must return an exception from this method. If no error occurred, this method must return null. The error must be set before this Export Registration is returned. Asynchronously occurring errors must be reported to the log.

Returns The exception that occurred during the initialization of this registration or null if no exception occurred.

122.12.8.3 public ExportReference getExportReference()

□ Return the Export Reference for the exported service.

Returns The Export Reference for this registration, or null if this Import Registration is closed.

Throws IllegalStateException– When this registration was not properly initialized. See getException().

122.12.8.4 public EndpointDescription update(Map<String, ?> properties)

properties properties to be merged with the current service properties for the ServiceReference represented by this ExportRegistration. If null is passed then the original properties passed to RemoteServiceAdmin.exportService(ServiceReference, Map) will be used.

□ Update the endpoint represented by this ExportRegistration and return an updated EndpointDescription. If this method returns an updated EndpointDescription, then the object returned via getExportReference() must also have been updated to return this new object. If this method does not return an updated EndpointDescription then the object returned via getExportReference() should remain unchanged. When creating the updated EndpointDescription the ServiceReference originally passed to RemoteServiceAdmin.exportService(ServiceReference, Map) must be queried to pick up any changes to its service properties. If this argument is null then the original properties passed when creating this ExportRegistration should be used when constructing the updated EndpointDescription. Otherwise the new properties should be used, and replace the original properties for subsequent calls to the update method.

Returns The updated EndpointDescription for this registration or null if there was a failure updating the endpoint. If a failure occurs then it can be accessed using getException().

Throws IllegalStateException– If this registration is closed, or when this registration was not properly initialized. See getException().

Since 1.1

122.12.9 public interface ImportReference

An Import Reference associates an active proxy service to a remote endpoint. The Import Reference can be used to reference an imported service. When the service is no longer imported, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.9.1 public EndpointDescription getImportedEndpoint()

□ Return the Endpoint Description for the remote endpoint.

Returns The Endpoint Description for the remote endpoint. Must be null when the service is no longer imported.

122.12.9.2 public ServiceReference<?> getImportedService()

□ Return the Service Reference for the proxy for the endpoint.

Returns The Service Reference to the proxy for the endpoint. Must be null when the service is no longer imported.

122.12.10 public interface ImportRegistration

An Import Registration associates an active proxy service to a remote endpoint. The Import Registration can be used to delete the proxy associated with an endpoint. It is created with the

RemoteServiceAdmin.importService(EndpointDescription) method. When this Import Registration has been closed, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.10.1 public void close()

□ Close this Import Registration. This must close the connection to the endpoint and unregister the proxy. After this method returns, all other methods must return null. This method has no effect when this registration has already been closed or is being closed.

122.12.10.2 public Throwable getException()

□ Return the exception for any error during the import process. If the Remote Service Admin for some reasons is unable to properly initialize this registration, then it must return an exception from this method. If no error occurred, this method must return null. The error must be set before this Import Registration is returned. Asynchronously occurring errors must be reported to the log.

Returns The exception that occurred during the initialization of this registration or null if no exception occurred.

122.12.10.3 public ImportReference getImportReference()

□ Return the Import Reference for the imported service.

Returns The Import Reference for this registration, or null if this Import Registration is closed.

Throws IllegalStateException– When this registration was not properly initialized. See getException().

122.12.10.4 public boolean update(EndpointDescription endpoint)

endpoint The updated endpoint

□ Update the local service represented by this ImportRegistration. After this method returns the EndpointDescription returned via getImportReference() must have been updated.

Returns true if the endpoint was successfully updated, false otherwise. If the update fails then the failure can be retrieved from getException().

Throws IllegalStateException– When this registration is closed, or if it was not properly initialized. See getException().

IllegalArgumentException– When the supplied EndpointDescription does not represent the same endpoint as this ImportRegistration.

Since 1.1

122.12.11 public class RemoteConstants

Provide the definition of the constants used in the Remote Service Admin specification.

Concurrency Immutable

122.12.11.1 public static final String ENDPOINT_FRAMEWORK_UUID = "endpoint.framework.uuid"

Endpoint property identifying the universally unique id of the exporting framework. Can be absent if the corresponding endpoint is not for an OSGi service.

The value of this property must be of type String.

122.12.11.2 public static final String ENDPOINT_ID = "endpoint.id"

Endpoint property identifying the id for this endpoint. This service property must always be set.

The value of this property must be of type String.

122.12.11.3 public static final String ENDPOINT_PACKAGE_VERSION_ = "endpoint.package.version."

Prefix for an endpoint property identifying the interface Java package version for an interface. For example, the property endpoint.package.version.com.acme=1.3 describes the version of the package for the com.acme.Foo interface. This endpoint property for an interface package does not have to be set. If not set, the value must be assumed to be 0.

Since endpoint properties are stored in a case insensitive map, case variants of a package name are folded together.

The value of properties having this prefix must be of type String.

122.12.11.4 public static final String ENDPOINT_SERVICE_ID = "endpoint.service.id"

Endpoint property identifying the service id of the exported service. Can be absent or 0 if the corresponding endpoint is not for an OSGi service.

The value of this property must be of type Long.

122.12.11.5 public static final String REMOTE_CONFIGS_SUPPORTED = "remote.configs.supported"

Service property identifying the configuration types supported by a distribution provider. Registered by the distribution provider on one of its services to indicate the supported configuration types.

The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.6 public static final String REMOTE_INTENTS_SUPPORTED = "remote.intents.supported"

Service property identifying the intents supported by a distribution provider. Registered by the distribution provider on one of its services to indicate the vocabulary of implemented intents.

The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.7 public static final String SERVICE_EXPORTED_CONFIGS = "service.exported.configs"

Service property identifying the configuration types that should be used to export the service. Each configuration type represents the configuration parameters for an endpoint. A distribution provider should create an endpoint for each configuration type that it supports.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.8 public static final String SERVICE_EXPORTED_INTENTS = "service.exported.intents"

Service property identifying the intents that the distribution provider must implement to distribute the service. Intents listed in this property are reserved for intents that are critical for the code to function correctly, for example, ordering of messages. These intents should not be configurable.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.9 public static final String SERVICE_EXPORTED_INTENTS_EXTRA = "service.exported.intents.extra"

Service property identifying the extra intents that the distribution provider must implement to distribute the service. This property is merged with the service.exported.intents property before the distribution provider interprets the listed intents; it has therefore the same semantics but the property should be configurable so the administrator can choose the intents based on the topology. Bundles should therefore make this property configurable, for example through the Configuration Admin service.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.10 public static final String SERVICE_EXPORTED_INTERFACES = "service.exported.interfaces"

Service property marking the service for export. It defines the interfaces under which this service can be exported. This list must be a subset of the types under which the service was registered. The single value of an asterisk ('*' \u002A) indicates all the interface types under which the service was registered excluding the non-interface types. It is strongly recommended to only export interface types and not concrete classes due to the complexity of creating proxies for some type of concrete classes.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.11 public static final String SERVICE_IMPORTED = "service.imported"

Service property identifying the service as imported. This service property must be set by a distribution provider to any value when it registers the endpoint proxy as an imported service. A bundle can use this property to filter out imported services.

The value of this property may be of any type.

See Also Remote Services Specification

122.12.11.12 public static final String SERVICE_IMPORTED_CONFIGS = "service.imported.configs"

Service property identifying the configuration types used to import the service. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework. If multiple configuration types are listed in this property, then they must be synonyms for exactly the same remote endpoint that is used to export this service.

The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification, SERVICE_EXPORTED_CONFIGS

122.12.11.13 public static final String SERVICE_INTENTS = "service.intents"

Service property identifying the intents that this service implement. This property has a dual purpose:

- A bundle can use this service property to notify the distribution provider that these intents are already implemented by the exported service object.
- A distribution provider must use this property to convey the combined intents of: The exporting service, and the intents that the exporting distribution provider adds, and the intents that the importing distribution provider adds.

To export a service, a distribution provider must expand any qualified intents. Both the exporting and importing distribution providers must recognize all intents before a service can be distributed. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.12 public interface RemoteServiceAdmin

A Remote Service Admin manages the import and export of services. A Distribution Provider can expose a control interface. This interface allows a Topology Manager to control the export and import of services. The API allows a Topology Manager to export a service, to import a service, and find out about the current imports and exports.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.12.1 public Collection<ExportRegistration> exportService(ServiceReference<?> reference, Map<String, ?> properties)

reference The Service Reference to export.

properties The properties to create a local Endpoint that can be implemented by this Remote Service Admin. If this is null, the Endpoint will be determined by the properties on the service. The properties are the same as given for an exported service. They override any properties in the specified Service Reference (case insensitive). The properties objectClass and service.id, in any case variant, are ignored. Those properties in the Service Reference cannot be overridden. This parameter can be null, this should be treated as an empty map.

□ Export a service to a given Endpoint. The Remote Service Admin must create an Endpoint from the given description that can be used by other Distribution Providers to connect to this Remote Service Admin and use the exported service. The property keys of a Service Reference are case insensitive while the property keys of the specified properties map are case sensitive. A property key in the specified properties map must therefore override any case variant property key in the properties of the specified Service Reference.

If the caller does not have the appropriate EndpointPermission[endpoint,EXPORT] for an Endpoint, and the Java Runtime Environment supports permissions, then the getException method on the corresponding returned ExportRegistration will return a SecurityException.

Returns A Collection of ExportRegistrations for the specified Service Reference and properties. Multiple Export Registrations may be returned because a single service can be exported to multiple Endpoints depending on the available configuration type properties and the intents that they support. The result is never null but may be empty if this Remove Service Admin does not recognize any of the configuration types, or if the Remote Service Admin cannot support the relevant intents.

Throws IllegalArgumentException– If any of the properties for this configuration type has a value that is not syntactically correct, or if the service properties and the overlaid properties do not contain a RemoteConstants.SERVICE_EXPORTED_INTERFACES entry. This means that implementations must not ignore invalid values for property names that they recognize.

122.12.12.2 public Collection<ExportReference> getExportedServices()

□ Return the currently active Export References.

If the caller does not have the appropriate EndpointPermission[endpoint,READ] for an Endpoint, and the Java Runtime Environment supports permissions, then returned collection will not contain a reference to the exported Endpoint.

Returns A Collection of ExportReferences that are currently active.

122.12.12.3 public Collection<ImportReference> getImportedEndpoints()

□ Return the currently active Import References.

If the caller does not have the appropriate EndpointPermission[endpoint,READ] for an Endpoint, and the Java Runtime Environment supports permissions, then returned collection will not contain a reference to the imported Endpoint.

Returns A Collection of ImportReferences that are currently active.

122.12.12.4 public ImportRegistration importService(EndpointDescription endpoint)

endpoint The Endpoint Description to be used for import.

□ Import a service from an Endpoint. The Remote Service Admin must use the given Endpoint to create a proxy. This method can return null if the service could not be imported.

Returns An Import Registration that combines the Endpoint Description and the Service Reference or null if the Endpoint could not be imported.

Throws SecurityException– If the caller does not have the appropriate EndpointPermission[endpoint,IMPORT] for the Endpoint, and the Java Runtime Environment supports permissions.

122.12.13 public class RemoteServiceAdminEvent

Provides the event information for a Remote Service Admin event.

Concurrency Immutable

122.12.13.1 public static final int EXPORT_ERROR = 6

A fatal exporting error occurred. The Export Registration has been closed.

122.12.13.2 public static final int EXPORT_REGISTRATION = 2

Add an export registration. The Remote Service Admin will send this event when it exports a service. When the RemoteServiceAdminListener service is registered, the Remote Service Admin must notify the listener of all existing Export Registrations.

122.12.13.3 public static final int EXPORT_UNREGISTRATION = 3

Remove an export registration. The Remote Service Admin will send this event when it removes the export of a service.

122.12.13.4 public static final int EXPORT_UPDATE = 10

Update an export registration. The Remote Service Admin will send this event when it exports a service.

Since 1.1

122.12.13.5 public static final int EXPORT_WARNING = 7

A problematic situation occurred, the export is still active.

122.12.13.6 public static final int IMPORT_ERROR = 5

A fatal importing error occurred. The Import Registration has been closed.

122.12.13.7 public static final int IMPORT_REGISTRATION = 1

Add an import registration. The Remote Service Admin will send this event when it imports a service. When the RemoteServiceAdminListener service is registered, the Remote Service Admin must notify the listener of all existing Import Registrations.

122.12.13.8 public static final int IMPORT_UNREGISTRATION = 4

Remove an import registration. The Remote Service Admin will send this event when it removes the import of a service.

122.12.13.9 public static final int IMPORT_UPDATE = 9

Update an import registration. The Remote Service Admin will send this event when it updates a service.

Since 1.1

122.12.13.10 public static final int IMPORT_WARNING = 8

A problematic situation occurred, the import is still active.

122.12.13.11 public RemoteServiceAdminEvent(int type, Bundle source, ExportReference exportReference, Throwable exception)

type The event type.

source The source bundle, must not be null.

exportReference The exportReference, can not be null.

exception Any exceptions encountered, can be null.

□ Create a Remote Service Admin Event for an export notification.

122.12.13.12 public RemoteServiceAdminEvent(int type, Bundle source, ImportReference importReference, Throwable exception)

type The event type.

source The source bundle, must not be null.

importReference The importReference, can not be null.

exception Any exceptions encountered, can be null.

□ Create a Remote Service Admin Event for an import notification.

122.12.13.13 public Throwable getException()

□ Return the exception for this event.

Returns The exception or null.

122.12.13.14 public ExportReference getExportReference()

□ Return the Export Reference for this event.

Returns The Export Reference or null.

122.12.13.15 public ImportReference getImportReference()

□ Return the Import Reference for this event.

Returns The Import Reference or null.

122.12.13.16 public Bundle getSource()

□ Return the bundle source of this event.

Returns The bundle source of this event.

122.12.13.17 public int getType()

□ Return the type of this event.

Returns The type of this event.

122.12.14 public interface RemoteServiceAdminListener

A RemoteServiceAdminEvent listener is notified synchronously of any export or import registrations and unregistrations.

If the Java Runtime Environment supports permissions, then filtering is done. RemoteServiceAdminEvent objects are only delivered to the listener if the bundle which defines the listener object's class has the appropriate EndpointPermission[endpoint,READ] for the endpoint referenced by the event.

See Also RemoteServiceAdminEvent

Concurrency Thread-safe

122.12.14.1 public void remoteAdminEvent(RemoteServiceAdminEvent event)

event The RemoteServiceAdminEvent object.

□ Receive notification of any export or import registrations and unregistrations as well as errors and warnings.

122.13 org.osgi.service.remoteserviceadmin.namespace

Remote Service Admin Namespaces Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

122.13.1 Summary

- DiscoveryNamespace - Remote Services Discovery Provider Capability and Requirement Namespace.
- DistributionNamespace - Remote Services Distribution Provider Capability and Requirement Namespace.
- TopologyNamespace - Remote Services Topology Manager Capability and Requirement Namespace.

122.13.2 public final class DiscoveryNamespace extends Namespace

Remote Services Discovery Provider Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency Immutable

122.13.2.1 public static final String CAPABILITY_PROTOCOLS_ATTRIBUTE = "protocols"

The capability attribute used to specify the discovery protocols supported by this discovery provider. The value of this attribute must be of type String or List<String>.

122.13.2.2 public static final String DISCOVERY_NAMESPACE = "osgi.remoteserviceadmin.discovery"

Namespace name for Remote Services discovery provider capabilities and requirements.

122.13.3 public final class DistributionNamespace extends Namespace

Remote Services Distribution Provider Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency Immutable

122.13.3.1 public static final String CAPABILITY_CONFIGS_ATTRIBUTE = "configs"

The capability attribute used to specify the config types supported by this distribution provider. The value of this attribute must be of type String or List<String>.

122.13.3.2 public static final String DISTRIBUTION_NAMESPACE = "osgi.remoteserviceadmin.distribution"

Namespace name for Remote Services distribution provider capabilities and requirements.

122.13.4 public final class TopologyNamespace extends Namespace

Remote Services Topology Manager Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency Immutable

122.13.4.1 public static final String CAPABILITY_POLICY_ATTRIBUTE = "policy"

The capability attribute used to specify the policy or policies supported by this topology manager. The value of this attribute must be of type `String` or `List<String>`. Policy names are typically implementation specific, however the Remote Services Specification defines the *promiscuous* and *fail-over* policies for common use cases.

122.13.4.2 public static final String FAIL_OVER_POLICY = "fail-over"

The attribute value for Topology managers with a fail-over policy

See Also TopologyNamespace.CAPABILITY_POLICY_ATTRIBUTE

122.13.4.3 public static final String PROMISCUOUS_POLICY = "promiscuous"

The attribute value for Topology managers with a promiscuous policy

See Also TopologyNamespace.CAPABILITY_POLICY_ATTRIBUTE

122.13.4.4 public static final String TOPOLOGY_NAMESPACE = "osgi.remoteserviceadmin.topology"

Namespace name for Remote Services topology manager capabilities and requirements.

122.14 References

[1] *OSGi Service Property Namespace*
https://docs.osgi.org/reference/service-property-namespace.html

[2] *UUIDs*
https://en.wikipedia.org/wiki/Universally_Unique_Identifier

[3] *Service Location Protocol (SLP)*
https://en.wikipedia.org/wiki/Service_Location_Protocol

[4] *JGroups*
http://www.jgroups.org/

[5] *UDDI*
https://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration

[6] *Service Component Architecture (SCA)*
https://www.osoa.org/

123 JTA Transaction Services Specification

Version 1.0

123.1 Introduction

Transactions are the key abstraction to provide reliability with large scale distributed systems and are a primary component of enterprise systems. This specification provides an OSGi service based design for the Java Transaction Architecture (JTA) Specification, which describes the standard transaction model for Java applications. Providing the JTA specification as a service based model enables the use of independent implementations. This JTA Transaction Services Specification provides a managed model, where an Application Container (such as the Java EE EJB container) manages the transaction and the enlistment of resources, and an unmanaged model, where each application is responsible for these tasks itself.

This specification provides a brief overview of JTA and then the use of it through 3 transaction services: User Transaction, Transaction Manager, and Transaction Synchronization.

This specification is based on [1] *Java Transaction API Specification 1.1*.

123.1.1 Essentials

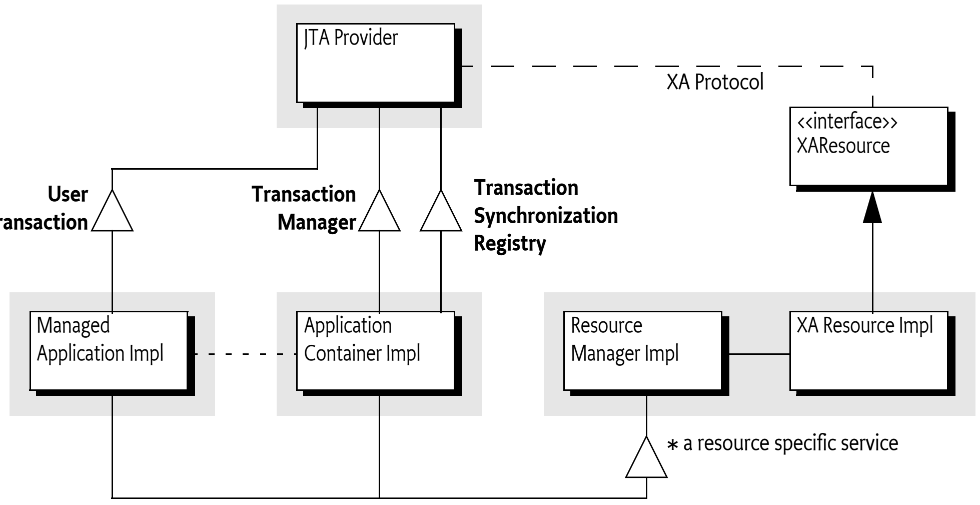
- *Portability* - It is important that applications are easy to port from other environments that support JTA.
- *Plugability* - Allow different vendors to provide implementations of this specification.
- *JTA Compatible* - Support full JTA 1.1 Specification

123.1.2 Entities

- *JTA Provider* - Implementation of this specification. It is responsible, on request from a Transaction Originator, for starting and ending transactions and coordinating the work of Resource Managers that become involved in each Transaction. This entity provides the User Transaction service, Transaction Manager service, and the Transaction Synchronization Registry service.
- *Transaction* - An atomic unit of work that is associated with a thread of execution.
- *Transaction Originator* - An Application or its Container, that directs the JTA Provider to begin and end Transactions.
- *User Transaction* - A service used by a Transaction Originator for beginning and ending transactions.
- *Transaction Manager* - A service used by a Transaction Originator for managing both transaction demarcation and enlistment of Durable Resources or Volatile Resources.
- *Transaction Synchronization Registry* - A service for enlistment of Volatile Resources for getting notifications before and after ending Transactions.
- *Application Bundle* - An entity that initiates work that executes under a Transaction.
- *Container* - An entity that is distinct from the Application and which provides a managed environment for Applications. Unmanaged environments do not distinguish between the Application and Container entities.

- *Resource Manager* - Provides the transactional resources whose work is externally coordinated by a JTA Provider. Examples of Resource Managers include databases, Java Message Service providers and enterprise information systems.
- *Durable Resource* - A resource whose work is made durable when the Transaction is successfully committed. Durable Resources can be enlisted with a Transaction to ensure that work is performed within the scope of the Transaction and to participate in the outcome of a Transaction. Durable Resource enlistment is the responsibility of the Application Bundle or its Container. Durable Resources implement the `javax.transaction.xa.XAResource` interface
- *Volatile Resource* - Resources that are associated with a Transaction but are no longer needed after the Transaction, for example transaction-scoped caches. Volatile Resources are registered with the JTA Provider to receive notifications before and after the outcome of the Transaction. Volatile Resources implement the `javax.transaction.Synchronization` interface
- *Transaction Services* - The triplet of the User Transaction, Transaction Manager, and Transaction Synchronization Registry services registered by the JTA Provider.

Figure 123.1 Transaction Service Specification Entities



123.1.3 Dependencies

This specification is based on the following packages:

```
javax.transaction
javax.transaction.xa
```

These packages must be exported as version 1.1.

123.1.4 Synopsis

The JTA Provider register the Transaction Services:

- *User Transaction* - Offers transaction demarcation capabilities to an Application bundle.
- *Transaction Manager* - Offers transaction demarcation and further transaction management capabilities to an Application Bundle or an Application Container.
- *Transaction Synchronization Registry* - Offers a callback registration service for volatile transactional participants wishing to be notified of the completion of the transaction.

A JTA Provider must register these services when it is started. A JTA Provider may put restrictions on which bundles can use these services. For example, in a Java EE environment, the JTA Provider does not expose the `TransactionManager` interface to applications. An OSGi environment which supports the Java EE specifications will typically provide access to the Transaction Manager service only to Java EE Containers.

A typical example of the use of a transaction is for transferring money from one bank account to another. Two Durable Resources are involved, one provided by the database from which the money is to be withdrawn and another provided by the database to which the money will be deposited. An Application Bundle acting as the Transaction Originator gets the User Transaction service and uses it to begin a transaction. This transaction is associated with the current thread (implicitly) by the JTA Provider. On the same thread of execution, the Application Bundle connects to the database from which the money is to be withdrawn and updates the balance in the source account by the amount to be debited.

The database is a resource manager whose connections have associated XA Resources; the first time a connection is used within the scope of a new transaction the Application Bundle, or a Container, obtains the XA Resource associated with the connection and enlists it with the JTA Provider through the Transaction Manager service. On the same thread of execution, the Application Bundle connects to the second database and updates the balance in the target account by the amount to be credited. An XA Resource for the second connection is enlisted with the Transaction Manager service as well by the Application Bundle or a Container.

Now that the money has been transferred the Transaction Originator requests a commit of the Transaction (on the same thread of execution) via the User Transaction Service, causing the JTA Provider to initiate the two-phase commit process with the two Resource Managers through the enlisted XA Resources. The transaction is then atomically committed or rolled back.

123.2 JTA Overview

A transaction is a unit of work in which interactions with multiple participants can be coordinated by a third party such that the final outcome of these interactions has well-defined transactional semantics. A variety of well-known transaction models exist with specific characteristics; the transactions described in this specification provide *Atomic Consistent Isolated and Durable* (ACID) semantics as defined in [2] *XA+ Specification* whereby all the participants in a transaction are coordinated to an *atomic* outcome in which the work of all the participants is either completely committed or completely rolled back.

The [2] *XA+ Specification* defines a *Distributed Transaction Processing* (DTP) software architecture for transactional work that is distributed across multiple Resource Managers and coordinated externally by a Transaction Manager using the two-phase commit XA protocol. The DTP architecture defines the roles of the *Transaction Manager* and *Resource Manager*; this specification uses the term *JTA Provider* rather than *Transaction Manager* to distinguish it from the *Transaction Manager service.* Note that Distributed Transaction Processing does not imply distribution of transactions across multiple frameworks or JVMs.

The [1] *Java Transaction API Specification 1.1* defines the Java interfaces required for the management of transactions on the enterprise Java platform.

123.2.1 Global and Local Transactions

A transaction may be a *local transaction* or a *global transaction.* A local transaction is a unit of work that is local to a single Resource Manager and may succeed or fail independently of the work of other Resource Managers. A global transaction, sometimes referred to as a distributed transaction, is a unit of work that may encompass multiple Resource Managers and is coordinated by a JTA Provider external to the Resource Manager(s) as described in the DTP architecture. The term *transaction* in this specification always refers to a global transaction.

The JTA Provider is responsible for servicing requests from a Transaction Originator to create and complete transactions, it manages the state of each transaction it creates, the association of each transaction with the thread of execution, and the coordination of any Resource Managers that become involved in the global transaction. The JTA Provider ensures that each transaction is associated with, at most, one application thread at a time and provides the means to move that association from one thread to another as needed.

The model for resource commit coordination is the *two phase commit* XA protocol, with Resource Managers being directed by the JTA Provider. The first time an Application accesses a Resource Manager within the scope of a new global transaction, the Application, or its Container, obtains an XA Resource from the Resource Manager and *enlists* this XA Resource with the JTA Provider.

At the end of a transaction, the Transaction Originator must decide whether to initiate a *commit* or *rollback* request for all the changes made within the scope of the Transaction. The Transaction Originator requests that the JTA Provider completes the transaction. The JTA Provider then negotiates with each enlisted Resource Manager to reach a coordinated outcome. A failure in the transaction at any point before the second phase of two-phase commit results in the transaction being rolled back.

XA is a *presumed abort* protocol and implementations of XA-compliant JTA Providers and Resource Managers can be highly optimized to perform no logging of transactional state until a commit decision is required. A Resource Manager durably records its prepare decision, and a JTA Provider durably records any commit decision it makes. Failures between a decision on the outcome of a transaction and the enactment of that outcome are handled during *transaction recovery* to ensure the atomic outcome of the transaction.

123.2.2 Durable Resource

Durable Resources are provided by Resource Managers and must implement the `XAResource` interface described in the [1] *Java Transaction API Specification 1.1*. An `XAResource` object is enlisted with a transaction to ensure that the work of the Resource Manager is associated with the correct transaction and to participate in the two-phase commit process. The `XAResource` interface is driven by the JTA Provider during the completion of the transaction and is used to direct the Resource Manager to commit or rollback any changes made under the corresponding transaction.

123.2.3 Volatile Resource

Volatile resources are components that do not participate in the two phase commit but are called immediately prior to and after the two phase commit. They implement the [1] *Java Transaction API Specification 1.1* `Synchronization` interface. If a request is made to commit a transaction then the volatile participants have the opportunity to perform some *before completion* processing such as flushing cached updates to persistent storage. Failures during the *before completion* processing must cause the transaction to rollback. In both the commit and rollback cases the volatile resources are called after two phase commit to perform *after completion* processing. *After completion* procession cannot affect the outcome of the transaction.

123.2.4 Threading

As noted above in *Global and Local Transactions* on page 543, a global transaction must not be associated with more than one application thread at a time but can be moved over time from one application thread to another. In some environments Applications run in containers which restrict the ability of the Application component to explicitly manage the transaction-thread association by restricting access to the Transaction Manager. For example, Java EE application servers provide web and EJB Containers for application components and, while the Containers themselves can explicitly manage transaction-thread associations, these containers do not allow the Applications to do so. Applications running in these containers are required to complete any transactions they start on that same application thread. In general, Applications that run inside a Container must follow the rules defined by that Container. For further details of the considerations specific to Java EE containers, see the section *Transactions and Threads* in [4] *Java Platform, Enterprise Edition (Java EE) Specification, v5*.

123.3 Application

An *Application* is a bundle that may use transactions, either as a Transaction Originator or as a bundle that is called as part of an existing transaction. A Transaction Originator Application bundle starts a transaction and end it with a commit or rollback using the User Transaction or Transaction Manager service.

A Transaction Originator Application bundle may not make use of Resource Managers itself but may simply provide transaction demarcation and then call other bundles which do use Resource Managers. In such a case the Transaction Originator Application bundle requires only the use of the User Transaction service for transaction demarcation. The called bundles may use the Transaction Manager service if they use Resource Managers.

Application Bundles that use Resource Managers have to know the enlistment strategy for the Resource Managers they use. There are two possibilities:

- *Application Bundle Enlistment* - The Application Bundle must enlist the Resource Managers itself. For each Resource Manager it uses it must enlist that Resource Manager with the Transaction Manager.
- *Container-Managed Enlistment* - An Application runs in a container, such as a Java EE Container, which manages the Resource Manager enlistment on behalf of the Application.

These scenarios are explained in the following sections.

123.3.1 No Enlistment

A Transaction Originator Application bundle that uses no Resource Managers itself but starts a Transaction before calling another bundle may use the *User Transaction* service to control the Transaction demarcation.

For example, an Application can use the User Transaction service to begin a global transaction:

```
UserTransaction ut = getUserTransaction();
ut.begin();
```

The User Transaction service associates a transaction with the current thread until that transaction is completed via:

```
UserTransaction ut = getUserTransaction();
ut.commit();
```

Or the equivalent `rollback` method. The `getUserTransaction` method implementation (not shown) can get the User Transaction service directly from the service registry or from an injected field.

123.3.2 Application Bundle Enlistment

An Application Bundle is responsible for enlisting Resource Managers itself. That is, it must enlist Resource Manager it uses with the *Transaction Manager* service. The Transaction Manager service is an implementation of the JTA `TransactionManager` interface, registered by the JTA Provider.

For example, an Application Bundle can get an `XADataSource` object from a Data Source Factory service. Such a Data Source object can provide an `XAConnection` object that then can provide an `XAResource` object. `XAResource` objects can then be enlisted with the Transaction Manager service.

For example:

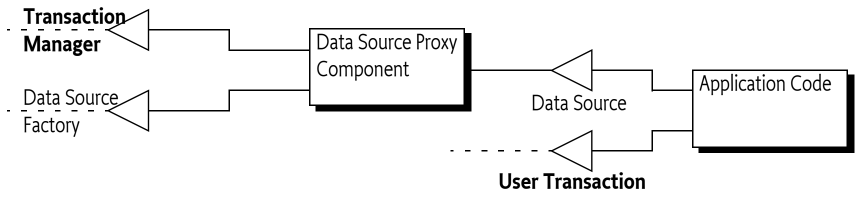
```
TransactionManager tm;
XADataSource       left;
XADataSource        right;

void acid() throws Exception {
   tm.begin();
    Transaction transaction = tm.getTransaction();
   try {
        XAConnection left = this.left.getXAConnection();
        XAConnection right = this.right.getXAConnection();
        transaction.enlistResource( left.getXAResource());
        transaction.enlistResource( right.getXAResource());
        doWork(left.getConnection(), right.getConnection());
        tm.commit();
    } catch( Throwable t ) {
        tm.rollback();
        throw t; } }
// ...
void setTransactionManager( TransactionManager tm ) { this.tm= tm; }
void setDataSourceFactory( DataSourceFactory dsf ) {
    left = dsf.createXADataSource( getLeftProperties() );
    right = dsf.createXADataSource( getRightProperties() );
}
```

In the previous example, the Transaction Manager service could have been injected with a component model like Declarative Services:

```
<reference interface="javax.transaction.TransactionManager"
    bind="setTransactionManager"/>
<reference name="dsf" interface="org.osgi.service.jdbc.DataSourceFactory"
    bind="setDataSourceFactory"/>
```

For example, it is possible to provide a Data Source service that provides automatic enlistment of the Connection as an XA Resource when one of its getConnection methods is called inside a transaction. The following code contains a Declarative Service component that implement this design. The component references a Transaction Manager service and a Data Source Factory service and provides a Data Source service that proxies an XA Data Source. Applications depend on the Data Source service, assuming that the Data Source service automatically enlists the connections it uses inside a transaction. See for an overview Figure 123.2 on page 546.

Figure 123.2 *Data Source Proxy*



This general purpose Data Source Proxy component can be fully configured by the Configuration Admin service to instantiate this component for each needed database connection. The Declarative Services service properties can be used to select a Data Source Factory for the required database driver (using the target), as well as provide the configuration properties for the creation of an XA Data Source. That is, such a component could be part of a support library.

The code for such an Application component could start like:

```
public class DataSourceProxy implements DataSource{
    Properties         properties  = new Properties();
    TransactionManager tm;
    XADataSource       xads;
```

The activate method is called when the component's dependencies are met, that is, there is a Transaction Manager service as well as a matching Data Source Factory service. In this method, the properties of the component are copied to a Properties object to be compatible with the Data Source Factory factory methods.

```
void activate(ComponentContext c) {
    // copy the properties set by the Config Admin into properties
    ...
}
```

The relevant methods in the Data Source Proxy component are the getConnection methods. The contract for this proxy component is that it enlists the XA Data Connection's XA Resource when it is called inside a transaction. This enlistment is done in the private enlist method.

```
public Connection getConnection() throws SQLException{
    XAConnection connection = xads.getXAConnection();
    return enlist(connection); }

public Connection getConnection(String username, String password)
        throws SQLException {
    XAConnection connection = xads.getXAConnection(username,password);
    return enlist(connection); }
```

The enlist method checks if there currently is a transaction active. If not, it ignores the enlistment, the connection will then not be connection to the transaction. If there is a current transaction, it enlists the corresponding XA Resource.

```
private Connection enlist(XAConnection connection)throws SQLException {
    try {
        Transaction transaction = tm.getTransaction();
        if (transaction != null)
            transaction.enlistResource( connection.getXAResource());
    } catch (Exception e) {
        SQLException sqle=
            new SQLException("Failed to enlist");
        sqle.initCause(e);
        throw sqle;
    }
    return connection.getConnection();
}
```

What remains are a number of boilerplate methods that forward to the XA Data Source or set the dependencies.

```
void setTransactionManager(TransactionManagertm) { this.tm = tm;}
void setDataSourceFactory(DataSourceFactory dsf) throws Exception{
    xads = dsf.createXADataSource(properties);}
public PrintWriter getLogWriter()
    throws SQLException { return xads.getLogWriter(); }

public int getLoginTimeout()
```

```
    throws SQLException { return xads.getLoginTimeout();}

public void setLogWriter(PrintWriter out)
    throws SQLException { xads.setLogWriter(out); }

public void setLoginTimeout(int seconds)
    throws SQLException { xads.setLoginTimeout(seconds);}
```

This is a fully coded example, it only lacks the configuration definitions for the Configuration Admin service.

This example Data Source proxy component makes it possible for an Application to depend on a Data Source service. The connections the Application uses from this Data Source are automatically transactional as long as there is a current transaction when the service is called. However, this approach only works when all bundles in the OSGi framework follow the same enlistment strategy because this specification does not provide a common enlistment strategy.

123.3.3 Container Managed Enlistment

The Application Container is responsible for enlisting Resource Managers used by the Application. For example, the Java EE Web and EJB Containers have a well defined model for managing resources within a transaction. If an Application runs inside a Java EE Container then it is the responsibility of the Java EE Container to handle the resource enlistment, the actual rules are beyond this specification.

A Transaction Originator Application bundle running inside a Container which manages any Resource Managers enlistment may use the User Transaction service for transaction demarcation, assuming this service is made available by the Container.

When a Java EE Container runs inside an OSGi framework then it must ensure that any services seen by its contained Applications are the same Transaction services as other bundles on that OSGi framework.

123.4 Resource Managers

Resource Managers perform work that needs to be committed or rolled back in a transaction. To participate in a transaction, a Resource Manager must have an XA Resource enlisted with the current transaction. This specification does not define how OSGi service implementations should be enlisted. This can be done by a Java EE Container, the Applications themselves, or through some other unspecified means.

123.5 The JTA Provider

The JTA Provider is the entity that provides the transaction services:

- *User Transaction - A service that implements the JTA* `UserTransaction` *interface.*
- *Transaction Manager - A service that implements the JTA* `TransactionManager` *interface.*
- *Transaction Synchronization Registry - A service that implements the JTA* `TransactionSynchronizationRegistry` *interface.*

There can be at most one JTA Provider in an OSGi framework and this JTA Provider must ensure that at most one transaction is associated with an application thread at any moment in time. All JTA Provider's transaction services must map to the same underlying JTA implementation. All JTA services should only be registered once.

123.5.1 User Transaction

The User Transaction service may be used by an Application bundle, acting as the Transaction Originator, to demarcate transaction boundaries when the bundle has no need to perform resource enlistment.

123.5.2 Transaction Manager

The Transaction Manager service offers transaction demarcation and further transaction management capabilities, such as Durable and Volatile resource enlistment, to an Application bundle or Application Container.

123.5.3 Transaction Synchronization Service

The Transaction Synchronization Registry service may be used by an Application bundle or a Container. The service provides for the registration of Volatile Resources that implement the JTA Synchronization interface.

For example:

```
private class MyVolatile implements Synchronization{...}
TransactionSynchronizationRegistry tsr = ...; // may be injected
tsr.registerInterposedSynchronization(new MyVolatile());
```

123.6 Life Cycle

123.6.1 JTA Provider

The life cycle of the transaction services and bundles that make up the JTA Provider must be dealt with appropriately such that implementations always ensure the atomic nature of transactions. When the JTA Provider is stopped and its services are unregistered, the JTA Provider must make sure that all active transactions are dealt with appropriately. A JTA Provider can decide to rollback all active transactions or it can decide to keep track of existing active transactions and allow them to continue to their normal conclusion but not allow any new transactions to be created. Any failures caused by executing code outside their life cycle can be dealt with as general failures. From a transactional consistency point of view, stopping the bundle(s) that implement the JTA Provider while transactional work is in-flight, is no different from a failure of the framework hosting the JTA Provider. In either case transaction recovery is initiated by the JTA Provider after it has re-started.

There are well-defined XA semantics between a JTA Provider and Resource Managers in the event of a failure of either at any point in a transaction. If a Resource Manager bundle is stopped while it is involved in-flight transactions then the JTA Provider should exhibit the same external behavior it does in the event of a communication failure with the Resource Manager. For example a JTA Provider will respond to an XAER_RMFAIL response resulting from calling the XAResource commit method by retrying the commit. The mechanism used by the JTA Provider to determine when to retry the commit is a detail of the implementation.

123.6.2 Application Bundles

Applications can act in the role of the Transaction Originator. There is no guarantee that an Application that starts a transaction will always be available to complete the transaction since the client can fail independently of the JTA Provider. A failure of the Application Bundle to complete, in a timely fashion, a transaction it originated must finally result in the JTA Provider rolling back the transaction.

123.6.3 Error Handling

This specification does not define a specific error handling strategy. Exceptions and errors that occur during transaction processing can result in the transaction being marked *rollback-only* by the container or framework in which an Application runs or may be left for the Application to handle. An Application which receives an error or an exception while running under a transaction can choose to mark the transaction rollback-only.

123.7 Security

This specification relies on the security model of JTA.

123.8 References

[1] *Java Transaction API Specification 1.1*
https://www.oracle.com/java/technologies/jta.html

[2] *XA+ Specification*
Version 2, The Open Group, ISBN: 1-85912-046-6

[3] *Transaction Processing*
J. Gray and A. Reuter. Morgan Kaufmann Publishers, ISBN 1.55860-190-2

[4] *Java Platform, Enterprise Edition (Java EE) Specification, v5*
https://jcp.org/en/jsr/detail?id=244

125 Data Service Specification for JDBC™ Technology

Version 1.1

125.1 Introduction

The Java Database Connectivity (JDBC) standard provides an API for applications to interact with relational database systems from different vendors. To abstract over concrete database systems and vendor specific characteristics, the JDBC specification provides various classes and Service Provider Interfaces (SPI) that can be used for database interaction. Implementations are database specific and provided by the corresponding driver. This specification defines how OSGi-aware JDBC drivers can provide access to their implementations. Applications can rely on this mechanism to transparently access drivers and to stay independent from driver specific classes. Additionally, this mechanism helps to use common OSGi practices and to avoid class loading problems.

This specification uses a number of packages that are defined in Java SE 1.4 or later.

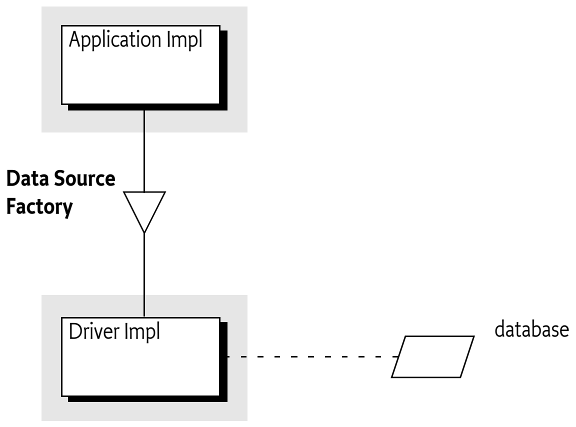
125.1.1 Essentials

- *Registration* - Provide a mechanism for JDBC driver announcements.
- *Lookup* - Inspect available database drivers and provide means for driver access.
- *Services* - Uses a service model for getting the driver objects.
- *Compatible* - Minimize the amount of work needed to support this specification for existing drivers.

125.1.2 Entities

- *Relational Database Management Systems* - (RDBMS) An external database system.
- *Database Driver* - JDBC-compliant database driver that is delivered in a bundle.
- *Data Source Factory* - Provides one of the different Data Sources that gives access to a database driver.
- *Application* - The application that wants to access a relational database system.

Figure 125.1 JDBC Class/Service Overview



125.1.3 Dependencies

The classes and interfaces used in this specification come from the following packages:

```
javax.sql
java.sql
```

These packages have no associated version. It is assumed they come from the runtime environment. This specification is based on Java SE 1.4 or later.

125.1.4 Synopsis

A JDBC *Database Driver* is the software that maps the JDBC specification to a specific implementation of a relational database. For OSGi, JDBC drivers are delivered as driver bundles. A driver bundle registers a Data Source Factory service when it is ACTIVE. Service properties are used to specify the database driver name, version, etc. The Data Source Factory service provides methods to create DataSource, ConnectionPoolDataSource, XADataSource, or Driver objects. These objects are then used by an application to interact with the relational database system in the standard way.

The application can query the service registry for available Data Source Factory services. It can select particular drivers by filtering on the service properties. This service based model is easy to use with dependency injection frameworks like Blueprint or Declarative Services.

125.2 Database Driver

A Database Driver provides the connection between an *Application* and a particular database. A single OSGi Framework can contain several Database Drivers simultaneously. To make itself available to Applications, a Database Driver must register a Data Source Factory service. Applications must be able to find the appropriate Database Driver. The Database Driver must therefore register the Data Source Factory service with the following service properties:

- OSGI_JDBC_DRIVER_CLASS - (String) The required name of the driver implementation class. This property is the primary key to find a driver's Data Source Factory. It is not required that there is an actual class with this name.
- OSGI_JDBC_DRIVER_NAME - (String) The optional driver name. This property is informational.
- OSGI_JDBC_DRIVER_VERSION - (String) The driver version. The version is not required to be an OSGi version, it should be treated as an opaque string. This version is likely not related to the package of the implementation class or its bundle.

The previous properties are vendor-specific and are meant to further describe the Database Driver to the Application.

Each Data Source Factory service must relate to a single Database Driver. The Database Driver implementation bundle does not necessarily need to be the registrar of the Data Source Factory service. Any bundle can provide the Data Source Factory service and delegate to the appropriate driver specific implementation classes. However, as JDBC driver implementations evolve to include built-in support for OSGi they can provide the Data Source Factory service themselves. This implies that the same driver can be registered multiple times.

125.2.1 Life Cycle

A Data Source Factory service should be registered while its Driver Bundle is in the ACTIVE state or when it has a lazy activation policy and is in the STARTING state.

What happens to the objects created by the Data Source Factory service, and the objects they created, is undefined in this specification. Database Drivers are not mandated to track the proper life cycle of these objects.

125.2.2 Package Dependencies

A Database Driver must import the javax.sql package. The java.sql package that contains the Driver and SQLException interface is automatically visible because it starts with java.. Both packages are contained in the JRE since Java SE 1.4. These packages are not normally versioned with OSGi version numbers. Bundles using the Data Source Factory must therefore ensure they get the proper imports, which is usually from the JRE. Due to the lack of specified metadata, the deployer is responsible for ensuring this.

125.3 Applications

125.3.1 Selecting the Data Source Factory Service

Applications can query the OSGi service registry for available Database Drivers by getting a list of Data Source Factory services. Normally, the application needs access to specific drivers that match their needed relational database type. The service properties can be used to find the desired Database Driver. This model is well supported by dependency injection frameworks like Blueprint or Declarative Services. However, it can of course also be used with the basic service methods. The following code shows how a Service Tracker can be used to get a Database Driver called ACME DB.

```
Filter filter = context.createFilter(
    "(&(objectClass=" +
        DataSourceFactory.class.getName() +
    ")(" +
        DataSourceFactory.OSGI_JDBC_DRIVER_CLASS + "=com.acme.db.Driver))");

ServiceTracker tracker = new ServiceTracker(context, filter, null);
tracker.open();

DataSourceFactory dsf = (DataSourceFactory) tracker.getService();
```

125.3.2 Using Database Drivers

The Data Source Factory service can be used to obtain instances for the following JDBC related types:

- javax.sql.DataSource
- javax.sql.ConnectionPoolDataSource

- javax.sql.XADataSource
- java.sql.Driver

Which type of Connection provider that is actually required depends on the Application and the use case. For each type, the Data Source Factory service provides a method that returns the corresponding instance. Each method takes a Properties object as a parameter to pass a configuration to the Database Driver implementation. The configuration is driver-specific and can be used to specify the URL for the database and user credentials. Common property names for these configuration properties are also defined in the DataSourceFactory interface.

A Data Source Factory service is not required to implement all of the factory methods. To indicate which factory method is implemented, the Data Source Factory service must be registered with the service property osgi.jdbc.datasourcefactory.capability having one or more of the following values.

- driver
- datasource
- connectionpooldatasource
- xadatasource

If an implementation does not support a particular type then it must throw a SQL Exception.

The following code shows how a DataSource object could be created.

```
Properties props = new Properties();
props.put(DataSourceFactory.JDBC_URL,      "jdbc:acme:ACMEDB");
props.put(DataSourceFactory.JDBC_USER,     "foo");
props.put(DataSourceFactory.JDBC_PASSWORD, "secret");
DataSource dataSource = dsf.createDataSource(props);
```

The DataSourceFactory interface has several static fields that represent common property keys for the Properties instance. General properties are:

- JDBC_DATABASE_NAME
- JDBC_DATASOURCE_NAME
- JDBC_DESCRIPTION
- JDBC_NETWORK_PROTOCOL
- JDBC_PASSWORD
- JDBC_PORT_NUMBER
- JDBC_ROLE_NAME
- JDBC_SERVER_NAME
- JDBC_USER
- JDBC_URL

The following additional property keys are provided for applications that want to create a ConnectionPoolDataSource object or a XAPoolDataSource object:

- JDBC_INITIAL_POOL_SIZE
- JDBC_MAX_IDLE_TIME
- JDBC_MAX_POOL_SIZE
- JDBC_MAX_STATEMENTS
- JDBC_MIN_POOL_SIZE
- JDBC_PROPERTY_CYCLE

Which property keys and values are supported depends on the driver implementation. Drivers can support additional custom configuration properties.

125.3.3 Using JDBC in OSGi and Containers

The JDBC service provides JDBC driver services, not *container* services. A typical client would only use the `DataSourceFactory.createDataSource()` method to procure a regular Data Source from which they can obtain (usually non-pooled) connections.

Containers generally offer connection pools and support XA transactions. The container manages the pools and does this by using Pooled Connection or XA Connection objects from a driver-implemented respective Connection Pool Data Source or XA Data Source. To support containers, frameworks, or any client that wants to manage a pool, these Data Source types are included in the Data Source Factory service. Drivers are permitted to implement their own Data Source using an underlying connection pooling scheme. This is driver-dependent and not related to the OSGi specifications.

The usual set of JDBC properties are defined in the services for use with the Data Source types. They are the same as what is defined for JDBC and the caller should know which properties make sense when passed to a given Data Source type. The same result should occur in OSGi as occurs outside of OSGi. If the driver does not support a given property with a given Data Source type then it can ignore it or it can throw an Exception.

125.4 Security

This specification depends on the JDBC specification for security.

125.5 org.osgi.service.jdbc

JDBC Service Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jdbc; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jdbc; version="[1.1,1.2)"
```

125.5.1 Summary

- `DataSourceFactory` - A factory for JDBC connection factories.

125.5.2 public interface DataSourceFactory

A factory for JDBC connection factories. There are 3 preferred connection factories for getting JDBC connections: `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource`. DataSource providers should implement this interface and register it as an OSGi service with the JDBC driver class name in the OSGI_JDBC_DRIVER_CLASS property.

Concurrency Thread-safe

125.5.2.1 public static final String JDBC_DATABASE_NAME = "databaseName"

The "databaseName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.2 public static final String JDBC_DATASOURCE_NAME = "dataSourceName"

The "dataSourceName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.3 public static final String JDBC_DESCRIPTION = "description"

The "description" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.4 public static final String JDBC_INITIAL_POOL_SIZE = "initialPoolSize"

The "initialPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.5 public static final String JDBC_MAX_IDLE_TIME = "maxIdleTime"

The "maxIdleTime" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.6 public static final String JDBC_MAX_POOL_SIZE = "maxPoolSize"

The "maxPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.7 public static final String JDBC_MAX_STATEMENTS = "maxStatements"

The "maxStatements" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.8 public static final String JDBC_MIN_POOL_SIZE = "minPoolSize"

The "minPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.9 public static final String JDBC_NETWORK_PROTOCOL = "networkProtocol"

The "networkProtocol" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.10 public static final String JDBC_PASSWORD = "password"

The "password" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.11 public static final String JDBC_PORT_NUMBER = "portNumber"

The "portNumber" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.12 public static final String JDBC_PROPERTY_CYCLE = "propertyCycle"

The "propertyCycle" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.13 public static final String JDBC_ROLE_NAME = "roleName"

The "roleName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.14 public static final String JDBC_SERVER_NAME = "serverName"

The "serverName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.15 public static final String JDBC_URL = "url"

The "url" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.16 public static final String JDBC_USER = "user"

The "user" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.17 public static final String OSGI_JDBC_CAPABILITY = "osgi.jdbc.datasourcefactory.capability"

Service property used by a DataSourceFactory to declare the capabilities when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the create method can be used. The type of this service property is String+

Since 1.1

125.5.2.18 public static final String OSGI_JDBC_CAPABILITY_CONNECTIONPOOLDATASOURCE = "connectionpooldatasource"

OSGI_JDBC_CAPABILITY service property which indicates that the DataSourceFactory is able to provide a ConnectionPoolDataSource invoking the createConnectionPoolDataSource(Properties props) method.

Since 1.1

125.5.2.19 public static final String OSGI_JDBC_CAPABILITY_DATASOURCE = "datasource"

OSGI_JDBC_CAPABILITY service property which indicates that the DataSourceFactory is able to provide a DataSource invoking the createDataSource(Properties props) method.

Since 1.1

125.5.2.20 public static final String OSGI_JDBC_CAPABILITY_DRIVER = "driver"

OSGI_JDBC_CAPABILITY service property which indicates that the DataSourceFactory is able to provide a Driver invoking the createDriver(Properties props) method.

Since 1.1

125.5.2.21 public static final String OSGI_JDBC_CAPABILITY_XADATASOURCE = "xadatasource"

OSGI_JDBC_CAPABILITY service property which indicates that the DataSourceFactory is able to provide a XADataSource invoking the createXADataSource(Properties props) method.

Since 1.1

125.5.2.22 public static final String OSGI_JDBC_DRIVER_CLASS = "osgi.jdbc.driver.class"

Service property used by a JDBC driver to declare the driver class when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.23 public static final String OSGI_JDBC_DRIVER_NAME = "osgi.jdbc.driver.name"

Service property used by a JDBC driver to declare the driver name when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.24 public static final String OSGI_JDBC_DRIVER_VERSION = "osgi.jdbc.driver.version"

Service property used by a JDBC driver to declare the driver version when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.25 public ConnectionPoolDataSource createConnectionPoolDataSource(Properties props) throws SQLException

props The properties used to configure the ConnectionPoolDataSource. null indicates no properties. If the property cannot be set on the ConnectionPoolDataSource being created then a SQLException must be thrown.

□ Create a new ConnectionPoolDataSource using the given properties.

Returns A configured ConnectionPoolDataSource.

Throws SQLException– If the ConnectionPoolDataSource cannot be created.

125.5.2.26 public DataSource createDataSource(Properties props) throws SQLException

props The properties used to configure the DataSource . null indicates no properties. If the property cannot be set on the DataSource being created then a SQLException must be thrown.

□ Create a new DataSource using the given properties.

Returns A configured DataSource.

Throws SQLException– If the DataSource cannot be created.

125.5.2.27 public Driver createDriver(Properties props) throws SQLException

props The properties used to configure the Driver. null indicates no properties. If the property cannot be set on the Driver being created then a SQLException must be thrown.

□ Create a new Driver using the given properties.

Returns A configured Driver.

Throws SQLException– If the Driver cannot be created.

125.5.2.28 public XADataSource createXADataSource(Properties props) throws SQLException

props The properties used to configure the XADataSource. null indicates no properties. If the property cannot be set on the XADataSource being created then a SQLException must be thrown.

□ Create a new XADataSource using the given properties.

Returns A configured XADataSource.

Throws SQLException– If the XADataSource cannot be created.

125.6 References

[1] *Java SE*
https://www.oracle.com/java/technologies/java-se-glance.html

125.7 Changes

- Added the service property osgi.jdbc.datasourcefactory.capability to DataSourceFactory.

126 JNDI Services Specification

Version 1.0

126.1 Introduction

Naming and directory services have long been useful tools in the building of software systems. The ability to use a programming interface to publish and consume objects can provide many benefits to any system. The Java Naming and Directory Interface (JNDI) is a registry technology in Java applications, both in the Java SE and Java EE space. JNDI provides a vendor-neutral set of APIs that allow clients to interact with a naming service from different vendors.

The JNDI as used in the Java SE environment relies on the class loading model provided by the JDK to find providers. By default, it attempts to load the JNDI provider class using the Thread Context Class Loader. In an OSGi environment, this type of Context creation is not desirable since it relies on the JNDI provider classes being visible to the JNDI client, or require it to set the Context Class Loader; in both cases breaking modularity. For modularity reasons, it is important that clients are not required to express a dependency on the implementation of services they use.

This specification will define how JNDI can be utilized from within an OSGi framework. The specification consists of three key parts:

- *OSGi Service Model* - How clients interact with JNDI when running inside an OSGi Framework.
- *JNDI Provider Model* - How JNDI providers can advertise their existence so they are available to OSGi and traditional clients.
- *Traditional Model* - How traditional JNDI applications and providers can continue to work in an OSGi Framework without needing to be rewritten when certain precautions are taken.

126.1.1 Essentials

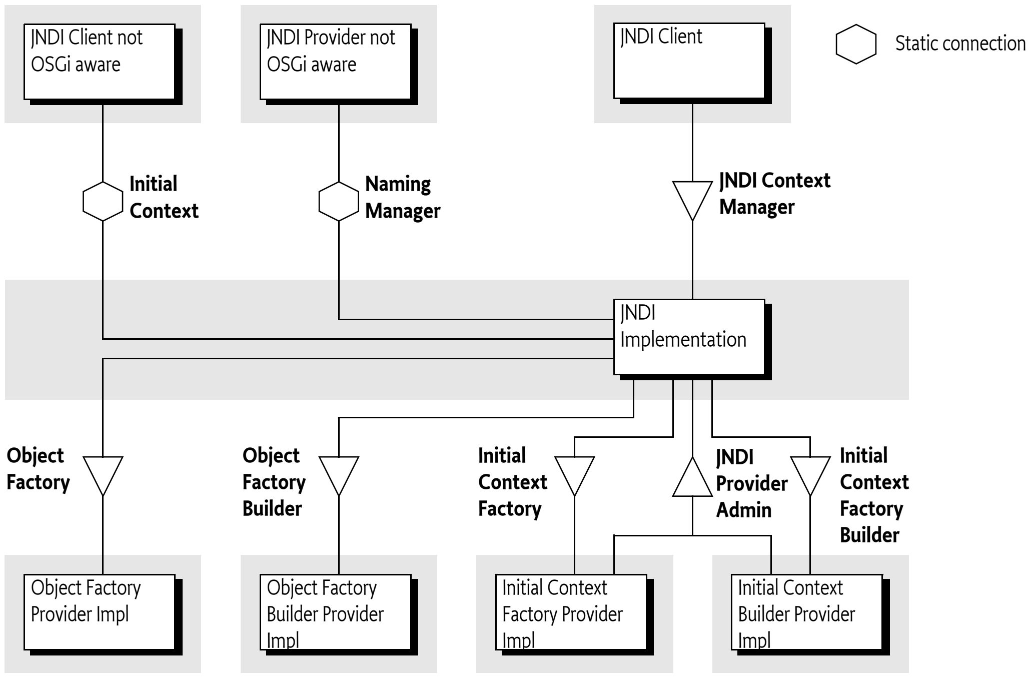
- *Naming Service* - Provide an integration model for JNDI API clients and providers.
- *Flexible* - Provide a standard mechanism for publishing and locating JNDI providers.
- *Compatibility* - Support the traditional JNDI programming model used by Java SE and Java EE clients.
- *Service Based* - Provide a service model that clients and providers can use to leverage JNDI facilities.
- *Migration* - Provide a mechanism to access OSGi services from a JNDI context.

126.1.2 Entities

- *JNDI Implementation* - The Implementer of the JNDI Context Manager, JNDI Provider Admin, and setter of the JNDI static singletons.
- *JNDI Client* - Any code running within an OSGi bundle that needs to use JNDI.
- *JNDI Context Manager* - A service that allows clients to obtain Contexts via a service.
- *JNDI Provider Admin* - A service that allows the conversion of objects for providers.
- *JNDI Provider* - Provides a Context implementation.
- *Context* - A Context abstracts a namespace. Implementations are provided by JNDI providers and the Contexts are used by JNDI clients. The corresponding interface is `javax.naming.Context`.

- *Dir Context* - A sub-type of Context that provides mechanisms for examining and updating the attributes of an object in a directory structure, and for performing searches in an hierarchical naming systems like LDAP. The corresponding interface is javax.naming.directory.DirContext.
- *Initial Context Factory* - A factory for creating instances of Context objects. This factory is used to integrate new JNDI Providers. In general, a single Initial Context Factory constructs Context objects for a single provider implementation. The corresponding interface is javax.naming.spi.InitialContextFactory.
- *Initial Context Factory Builder* - A factory for InitialContextFactory objects. A single Initial Context Factory Builder can construct InitialContextFactory objects for different types of Contexts. The interface is javax.naming.spi.InitialContextFactoryBuilder.
- *Object Factory* - Used in conversion of objects. The corresponding interface is javax.naming.spi.ObjectFactory.
- *Dir Object Factory* - An Object Factory that takes attribute information for object conversion. The corresponding interface is javax.naming.spi.DirObjectFactory.
- *Object Factory Builder* - A factory for ObjectFactory objects. A single Object Factory Builder can construct ObjectFactory instances for different types of conversions. The corresponding interface is javax.naming.spi.ObjectFactoryBuilder.
- *Reference* - A description of an object that can be turned into an object through an Object Factory. The associated Referenceable interface implemented on an object indicates that it can provide a Reference object.

Figure 126.1 JNDI Service Specification Service Entities


JNDI Client not OSGi aware
JNDI Provider not OSGi aware
JNDI Client
Static connection
Initial Context
Naming Manager
JNDI Context Manager
JNDI Implementation
Object Factory
Object Factory Builder
Initial Context Factory
JNDI Provider Admin
Initial Context Factory Builder
Object Factory Provider Impl
Object Factory Builder Provider Impl
Initial Context Factory Provider Impl
Initial Context Builder Provider Impl

126.1.3 Dependencies

The classes and interfaces used in this specification come from the following packages:

```
javax.naming
javax.naming.spi
javax.naming.directory
```

These packages have no associated version. It is assumed they come from the runtime environment. This specification is based on Java SE 1.4 or later.

126.1.4 Synopsis

A client bundle wishing to make use of JNDI in order to access JNDI Providers such as LDAP or DNS in OSGi should not use the Naming Manager but instead use the JNDI Context Manager service. This service can be asked for a Context based on environment properties. The environment properties are based on an optional argument in the newInitialContext method, the Java System properties, and an optional resource in the caller's bundle.

These environment properties can specify an implementation class name for a factory that can create a Context object. If such a class name is specified, then it is searched for in the service registry. If such a service is found, then that service is used to create a new Context, which is subsequently returned. If no class name is specified, the service registry is searched for Initial Context Factory services. These services are tried in ranking order, as specified in ServiceReference.compareTo, to see if they can create an appropriate Context, the first one that can create a Context is then used.

If no class name is specified, all Initial Context Factory Builder services are tried to see if they can create a Context, the first non-null result is used. If no Context can be found, a No Initial Context Exception is thrown. Otherwise, the JNDI Context Manager service returns an initial Context that uses the just created Context from a provider as the backing service. This initial Context delegates all operations to this backing Context, except operations that use a name that can be interpreted as a URL, that is, the name contains a colon. URL operations are delegated a URL Context that is associated with the used scheme. URL Contexts are found through the general object conversion facility provided by the JNDI Provider Admin service.

The JNDI Provider Admin service provides a general object conversion facility that can be extended with Object Factory and Object Factory Builder services that are traditionally provided through the Naming Manager getObjectInstance method. A specific case for this conversion is the use of Reference objects. Reference objects can be used to store objects persistently in a Context implementation. Reference objects must be converted to their corresponding object when retrieved from a Context.

During the client's use of a Context it is possible that its provider's service is unregistered. In this case the JNDI Context Manager must release the backing Context. If the initial Context is used and no backing Context is available, the JNDI Context Manager must re-create a new Context, if possible. Otherwise a Naming Exception is thrown. If subsequently a proper new backing Context can be created, the initial Context must start operating again.

The JNDI Context Manager service must track the life cycle of a calling bundle and ensure that any returned Context objects are closed and returned objects are properly cleaned up when the bundle is closed or the JNDI Context Manager service is unget.

When the client bundle is stopped, any returned initial Context objects are closed and discarded. If the Initial Context Factory, or Initial Context Factory Builder, service that created the initial Context goes away then the JNDI Context Manager service releases the Context backing the initial Context and attempts to create a replacement Context.

Clients and JNDI Context providers that are unaware of OSGi use static methods to connect to the JRE JNDI implementation. The InitialContext class provides access to a Context from a provider and providers use the static NamingManager methods to do object conversion and find URL Contexts. This traditional model is not aware of OSGi and can therefore only be used reliably if the consequences of this lack of OSGi awareness are managed.

126.2 JNDI Overview

The Java Naming and Directory Interface (JNDI) provides an abstraction for namespaces that is included in Java SE. This section describes the basic concepts of JNDI as provided in Java SE. These concepts are later used in the service model provided by this specification.

126.2.1 Context and Dir Context

The *[1] Java Naming and Directory Interface* (JNDI) defines an API for *namespaces*. These namespaces are abstracted with the `Context` interface. Namespaces that support *attributes*, such as a namespace as the Lightweight Directory Access Protocol (LDAP), are represented by the `DirContext` class, which extends the `Context` class. If applicable, a `Context` object can be cast to a `DirContext` object. The distinction is not relevant for this specification, except in places where it is especially mentioned.

The `Context` interface models a set of name-to-object *bindings* within a namespace. These bindings can be looked-up, created, and updated through the `Context` interface. The `Context` interface can be used for federated, flat, or hierarchical namespaces.

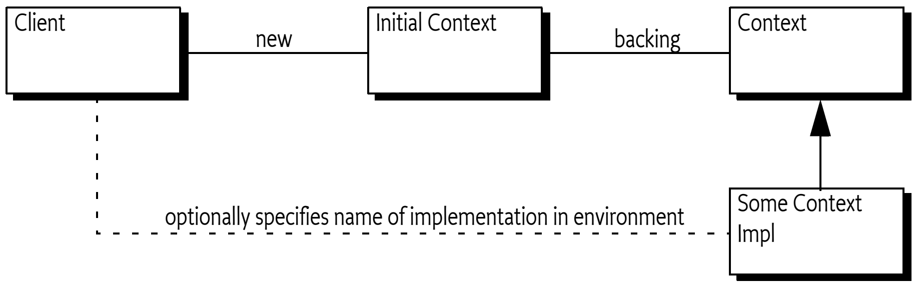
126.2.2 Initial Context

Obtaining a Context for a specific namespace, for example DNS, is handled through the `InitialContext` class. Creating an instance of this class will cause the JRE to find a *backing* Context. The Initial Context is only a facade for the backing Context. The facade context provides URL based lookups.

The backing Context is created by a *JNDI Provider*. How this backing Context is created is an elaborate process using class loading techniques or a provisioning mechanism involving *builders*, see *Naming Manager Singletons* on page 563 for more information about the builder provisioning mechanism.

If there is no Initial Context Factory Builder set, the class name of a class implementing the `InitialContextFactory` interface is specified as a property in the *environment*. The environment is a `Hashtable` object that is constructed from different sources and then merged with System properties and a resource in the calling bundle, see *Environment* on page 563. In a standard Java SE JNDI, the given class name is then used to construct an `InitialContextFactory` object and this object is then used to create the backing Context. This process is depicted in Figure 126.2 on page 562.

Figure 126.2 Backing Context



126.2.3 URL Context Factory

The `InitialContext` class implements the `Context` interface. It can therefore delegate all the `Context` interface methods to the backing `Context` object. However, it provides a special URL lookup behavior for names that are formed like URLs, that is, names that contain a colon (':' `\u003A`) character. This behavior is called a *URL lookup*.

URL lookups are not delegated to the backing Context but are instead first tried via a *URL Context* based lookup on the given scheme, like:

```
myscheme:foo
```

For example a lookup using acme:foo/javax.sql.DataSource results in a URL Context being used, rather than the backing Context.

JNDI uses class loading techniques to search for an ObjectFactory class that can be used to create this URL Context. The Naming Manager provides a static method getURLContext for this purpose. If such a URL Context is found, it is used with the requested operation and uses the full URL. If no such URL Context can be found, the backing Context is asked to perform the operation with the given name.

The URL lookup behavior is only done when the backing Context was created by the JNDI implementation in the JRE. If the backing Context had been created through the singleton provisioning mechanism, then no URL lookup is done for names that have a colon. The URL lookup responsibility is then left to the backing Context implementation.

126.2.4 Object and Reference Conversion

The NamingManager class provides a way to create objects from a *description* with the getObjectInstance method. In general, it will iterate over a number of ObjectFactory objects and ask each one of them to provide the requested object. The first non-null result indicates success. These ObjectFactory objects are created from an environment property.

A special case for the description argument in the getObjectInstance method is the *Reference*. A Reference is a description of an object that can be stored persistently. It can be re-created into an actual object through the static getObjectInstance method of the NamingManager class. The Reference object describes the actual ObjectFactory implementing class that must be used to create the object.

This default behavior is completely replaced with the Object Factory Builder singleton by getting the to be used ObjectFactory object directly from the set singleton Object Factory Builder.

126.2.5 Environment

JNDI clients need a way to set the configuration properties to select the proper JNDI Provider. For example, a JNDI Provider might require an identity and a password in order to access the service. This type of configuration is referred to as the *environment* of a Context. The environment is a set of properties. Common property names can be found in [3] *JNDI Standard Property Names*. The set of properties is build from the following sources (in priority order, that is later entries are shadowed by earlier entries):

1. Properties set in the environment Hashtable object given in the constructor argument (if any) of the InitialContext class.
2. Properties from the Java System Properties
3. Properties found in $JAVA_HOME/lib/jndi.properties

There are some special rules around the handling of specific properties.

126.2.6 Naming Manager Singletons

The default behavior of the JRE implementation of JNDI can be extended in a standardized way. The NamingManager class has two static singletons that allow JNDI Providers outside the JRE to provide InitialContextFactory and ObjectFactory objects. These singletons are set with the following static methods on the NamingManager class:

- setObjectFactoryBuilder(ObjectFactoryBuilder) - A hook to provide ObjectFactory objects.
- setInitialContextFactoryBuilder(InitialContextFactoryBuilder) - A hook to provide InitialContextFactory objects. This hook is consulted to create a Context object that will be associated with an InitialContext object the client creates.

These JNDI Provider hooks are *singletons* and must be set *before* any application code creates an InitialContext object or any objects are converted. If these singletons are not set, the JNDI implementation in the JRE will provide a default behavior that is based on searching through classes defined in an environment property.

Both singletons can only be set once. A second attempt to set these singletons results in an Illegal State Exception being thrown.

126.2.7 Built-In JNDI Providers

The Java Runtime Environment (JRE) defines the following default providers:

- *LDAP* - Lightweight Directory Access Protocol (LDAP) service provider
- *COS* - CORBA Object Service (COS) naming service provider
- *RMI* - Remote Method Invocation (RMI) Registry service provider
- *DNS* - Domain Name System (DNS) service provider

Although these are the default JNDI Service Providers, the JNDI architecture provides a number of mechanisms to plug-in new types of providers.

126.3 JNDI Context Manager Service

The JNDI Context Manager service allows clients to obtain a Context using the OSGi service model. By obtaining a JNDI Context Manager service, a client can get a Context object so that it can interact with the available JNDI Providers. This service replaces the approach where the creation of a new InitialContext object provided the client with access to an InitialContext object that was backed by a JNDI Provider's Context.

The JNDIContextManager interface defines the following methods for obtaining Context objects:

- newInitialContext() - Obtain a Context object using the default environment properties.
- newInitialContext(Map) - Get a Context object using the default environment properties merged with the given properties.
- newInitialDirContext() - Get a DirContext object using a default environment properties.
- newInitialDirContext(Map) -Get a DirContext object using the default environment properties merged with the given properties.

The JNDI Context Manager service returns Context objects that implement the same behavior as the InitialContext class; the returned Context object does not actually extend the InitialContext class, its only guarantee is that it implements the Context interface.

This Context object is a facade for the context that is created by the JNDI Provider. This JNDI Provider's Context is called the *backing Context*. This is similar to the behavior of the InitialContext class. However, in this specification, the facade can change or loose the backing Context due to the dynamics of the OSGi framework.

The returned facade must also provides URL lookups, just like an Initial Context. However, the URL Context lookup must be based on Object Factory services with a service property that defines the scheme.

The environment properties used to create the backing Context are constructed in a similar way as the environment properties of the Java SE JNDI, see *Environment and Bundles* on page 565.

The following sections define in detail how a JNDI Provider Context must be created and managed.

126.3.1 Environment and Bundles

The Java SE JNDI looks for a file in $JAVAHOME/lib/jndi.properties, see *Environment* on page 563. A JNDI Implementation must not use this information but it must use a resource in the bundle that uses the JNDI Context Manager service. The order is therefore:

1. Properties set in the environment Hashtable object given in the constructor argument (if any) of the InitialContext class.
2. Properties from the Java System Properties
3. A properties resource from the bundle that uses the service called /jndi.properties.

The following four properties do not overwrite other properties but are merged:

- java.naming.factory.object
- java.naming.factory.state
- java.naming.factory.control
- java.naming.factory.url.pkgs

These property values are considered lists and the ultimate value used by the JNDI Providers is taken by merging the values found in each stage into a single colon separated list. For more information see [3] *JNDI Standard Property Names.*

The environment consists of the merged properties. This environment is then passed to the Initial Context Factory Builder for the creation of an Initial Context Factory.

126.3.2 Context Creation

When a client calls one of the newInitialContext (or newInitialDirContext) methods, the JNDI Context Manager service must construct an object that implements the Context interface based on the environment properties. All factory methods in the InitialContextFactory and InitialContextFactoryBuilder classes take a Hashtable object with the environment as an argument, see *Environment and Bundles* on page 565.

The caller normally provides a specific property in the environment that specifies the class name of a provider class. This property is named:

```
java.naming.factory.initial
```

The algorithm to find the provider of the requested Context can differ depending on the presence or absence of the java.naming.factory.initial property in the environment.

In the following sections the cases for presence or absence of the java.naming.factory.initial property are described. Several steps in these algorithm iterate over a set of available services. This iteration must always take place in ranking order as specified in ServiceReference.compareTo.

Exception handling in the following steps is as follows:

- If an Exception is thrown by an Initial Context Factory Builder service, then this Exception must be logged but further ignored.
- Exceptions thrown by the InitialContextFactory objects when creating a Context must be thrown to the caller.

126.3.2.1 Implementation Class Present in Environment

If the implementation class is specified, a JNDI Provider is searched in the service registry with the following steps, which stop when a backing Context can be created:

1. Find a service in ranking order that has a name matching the given implementation class name as well as the InitialContextFactory class name. The searching must take place through the Bundle Context of the requesting bundle but must not require that the requesting bundle imports

the package of the implementation class. If such a matching Initial Context Factory service is found, it must be used to construct the Context object that will act as the backing Context.

2. Get all the Initial Context Factory Builder services. For each such service, in ranking order:
   - Ask the Initial Context Factory Builder service to create a new InitialContextFactory object. If this is null then continue with the next service.
   - Create the Context with the found Initial Context Factory and return it.
3. If no backing Context could be found using these steps, then the JNDI Context Manager service must throw a No Initial Context Exception.

126.3.2.2 No Implementation Class Specified

If the environment does not contain a value for the java.naming.factory.initial property then the following steps must be used to find a backing Context object.

1. Get all the Initial Context Factory Builder services. For each such service, in ranking order, do:
   - Ask the Initial Context Factory Builder service to create a new InitialContextFactory object. If this is null, then continue with the next service.
   - Create the backing Context object with the found Initial Context Factory service and return it.
2. Get all the Initial Context Factory services. For each such service, in ranking order, do:
   - Ask the Initial Context Factory service to create a new Context object. If this is null then continue with the next service otherwise create a new Context with the created Context as the backing Context.
3. If no Context has been found, an initial Context is returned without any backing. This returned initial Context can then only be used to perform URL based lookups.

126.3.3 Rebinding

A JNDI Provider can be added or removed to the service registry at any time because it is an OSGi service; OSGi services are by their nature dynamic. When a JNDI Provider unregisters an Initial Context Factory that was used to create a backing service then the JNDI Context Manager service must remove the association between any returned Contexts and their now invalid backing Contexts.

The JNDI Context Manager service must try to find a replacement whenever it is accessed and no backing Context is available. However, if no such replacement can be found the called function must result in throwing a No Initial Context Exception.

126.3.4 Life Cycle and Dynamism

When a client has finished with a Context object, then the client must close this Context object by calling the close method. When a Context object is closed, the resources held by the JNDI Implementation on the client's behalf for that Context must all be released. Releasing these resources must not affect other, independent, Context objects returned to the same client.

If a client ungets the JNDI Context Manager service, all the Context objects returned through that service instance must automatically be closed by the JNDI Context Manager. When the JNDI Context Manager service is unregistered, the JNDI Context Manager must automatically close all Contexts held.

For more information about life cycle issues, see also *Life Cycle Mismatch* on page 573.

126.4 JNDI Provider Admin service

JNDI provides a general object conversion service, see *Object and Reference Conversion* on page 563. For this specification, the responsibility of the static method on the NamingManager getObjectInstance is replaced with the JNDI Provider Admin service. The JNDIProviderAdmin interface provides the following methods that can be used to convert a description object to an object:

- getObjectInstance(Object,Name,Context,Map) - Used by Context implementations to convert a description object to another object.
- getObjectInstance(Object,Name,Context,Map,Attributes) - Used by a Dir Context implementations to convert a description object to another object.

In either case, the first argument is an object, called the *description*. JNDI allows a number of different Java types here. When either method is called, the following algorithm is followed to find a matching Object Factory to find/create the requested object. This algorithm is identical for both methods, except that the call that takes the Attributes argument consults Dir Object Factory services first and then Object Factory services while the method without the Attributes parameter only consults Object Factory services.

1. If the description object is an instance of Referenceable, then get the corresponding Reference object and use this as the description object.
2. If the description object is not a Reference object then goto step 5.
3. If a factory class name is specified, the JNDI Provider Admin service uses its own Bundle Context to search for a service registered under the Reference's factory class name. If a matching Object Factory is found then it is used to create the object from the Reference object and the algorithm stops here.
4. If no factory class name is specified, iterate over all the Reference object's StringRefAddrs objects with the address type of URL. For each matching address type, use the value to find a matching URL Context, see *URL Context Provider* on page 569, and use it to recreate the object. See the Naming Manager for details. If an object is created then it is returned and the algorithm stops here.
5. Iterate over the Object Factory Builder services in ranking order. Attempt to use each such service to create an ObjectFactory or DirObjectFactory instance. If this succeeds (non null) then use this ObjectFactory or DirObjectFactory instance to recreate the object. If successful, the algorithm stops here.
6. If the description was a Reference and without a factory class name specified, or if the description was not of type Reference, then attempt to convert the object with each Object Factory service (or Dir Object Factory service for directories) service in ranking order until a non-null value is returned.
7. If no ObjectFactory implementations can be located to resolve the given description object, the description object is returned.

If an Exception occurs during the use of an Object Factory Builder service then this exception should be logged but must be ignored. If, however, an Exception occurs during the calling of a found ObjectFactory or DirObjecFactory object then this Exception must be re-thrown to the caller of the JNDI Provider Admin service.

126.5 JNDI Providers

JNDI Providers can be registered by registering an appropriate service. These services are consulted by the JNDI Implementation for creating a Context as well as creating/finding/converting general objects.

126.5.1 Initial Context Factory Builder Provider

An Initial Context Factory Builder provider is asked to provide an Initial Context Factory when no implementation class is specified or no such implementation can be found. An Initial Context Factory Builder service can be used by containers for other bundles to control the initial Context their applications receive.

An Initial Context Factory Builder provider must register an Initial Context Factory Builder service. The iteration ordering of multiple Initial Context Factory Builder services must always take place in ranking order as specified in ServiceReference.compareTo. Implementations must be careful to correctly provide defaults.

For example, a container could use a thread local variable to mark the stack for a specific application. The implementation of the Initial Context Factory Builder can then detect specific calls from this application. To make the next code example work, an instance must be registered as an Initial Context Factory Builder service.

```
public class Container implements InitialContextFactoryBuilder {
    ThreadLocal<Application> apps;

    void startApp(final Application app) {
        Thread appThread = new Thread(app.getName()) {
            public void run() {
                apps.set(app);
                    app.run();
    }}}

    public InitialContextFactory
        createInitialContextFactory( Hashtable<?,?> ht ) {
        final Application app = apps.get();
        if ( app == null )
          return null;

        return new InitialContextFactory() {
            public Context getInitialContext( Hashtable<?,?>env) {
                return app.getContext(env);
            }
        };
 } }
```

126.5.2 Initial Context Factory Provider

An Initial Context Factory provides Contexts of a specific type. For example, those contexts allow communications with an LDAP server. An Initial Context Factory Provider must register the its Initial Context Factory service under the following names:

- *Implementation Class* - An Initial Context Factory provider must register a service under the name of the implementation class. This allows the JNDI Context Manager to find implementations specified in the environment properties.
- *Initial Context Factory* - As a general Initial Context Factory. If registered as such, it can be consulted for a default Initial Context. Implementations must be careful to only return a Context when the environment properties are appropriate. See *No Implementation Class Specified* on page 566

An Initial Context Factory service can create both DirContext as well as Context objects.

For example, SUN JREs for Java SE provide an implementation of a Context that can answer DNS questions. The name of the implementation class is a well known constant. The following class can be used with Declarative Services to provide a lazy implementation of a DNS Context:

```
public class DNSProvider implements InitialContextFactory{
    public Context createInitialContextFactory( Hashtable<?,?>env ) throws
```

```
        NamingException {
        try {
            Class<InitialContextFactory> cf = (Class<InitialContextFactory>)
                l.loadClass("com.sun.jndi.dns.DnsContextFactory" );
            InitialContextFactory icf = cf.newInstance();
            return icf.createInitialContextFactory(env);
        } catch( Throwable t ) {
            return null;
        }
    }
}
```

126.5.3 Object Factory Builder Provider

An Object Factory Builder provider must register an Object Factory Builder service. Such a service can be used to provide ObjectFactory and/or DirObjectFactory objects. An Object Factory Builder service is requested for such an object when no specific converter can be found. This service can be leveraged by bundles that act as a container for other bundles to control the object conversion for their subjects.

126.5.4 Object Factory Provider

An Object Factory provider can participate in the conversion of objects. It must register a service under the following names:

- *Implementation Class* - A service registered under its implementation class can be leveraged by a description that is a Reference object. Such an object can contain the name of the factory class. The implementation class can implement the DirObjectFactory interface or the ObjectFactory interface.
- *Object Factory* - The ObjectFactory interface is necessary to ensure class space consistency.
- *Dir Object Factory* - If the Object Factory provider can accept the additional Attributes argument in the getObjectInstance method of the JNDI Provider Admin service than it must also register as a Dir Object Factory service.

126.5.5 URL Context Provider

A *URL Context Factory* is a special type of an Object Factory service. A URL Context Factory must be registered as an Object Factory service with the following service property:

- osgi.jndi.url.scheme - The URL scheme associated with this URL Context, for example acme. The scheme must not contain the colon (':' \u003A).

A URL Context is used for URL based operations on an initial Context. For example, a lookup to acme:foo/javax.sql.DataSource must not use the provider based lookup mechanism of the backing Context but instead causes a lookup for the requested URL Context. A URL Context also provides a secondary mechanism for restoring Reference objects.

When an initial Context returned by the JNDI Context Manager service is given a URL based operation, it searches in the service registry for an Object Factory service that is published with the URL scheme property that matches the scheme used from the lookup request.

It then calls the getInstance method on the Object Factory service with the following parameters:

- *Object* - Should be either a String, String[], or null.
- *Name* - must be null
- *Context* - must be null
- *Hashtable* - The environment properties.

Calling the getInstance method must return a Context object. This context is then used to perform the lookup.

The life cycle of the Object Factory used to create the URL Context is tied to the JNDI context that was used to perform the URL based JNDI operation. By the time JNDI context is closed any ObjectFactory objects held to process the URL lookups must be released (unget).

126.5.6 JRE Context Providers

The Java Runtime Environment (JRE) defines a number of default naming providers., see *Built-In JNDI Providers* on page 564. These naming providers are not OSGi aware, but are commonly used and are provided by the JRE. These naming providers rely on the NamingManager class for object conversion and finding URL Contexts.

The JRE default providers are made available by the JNDI Implementation. This JNDI Implementation must register a *built-in* Initial Context Factory Builder service that is capable of loading any InitialContextFactory classes of the JRE providers.

When this built-in Initial Context Factory Builder is called to create an InitialContextFactory object it must look in the environment properties that were given as an argument and extract the java.naming.factory.initial property; this property contains the name of the class of a provider. The built-in Initial Context Factory Builder then must use the bootstrap class loader to load the given InitialContextFactory class and creates a new instance with the no arguments constructor and return it. If this fails, it must return null. This mechanism will allow loading of any built-in providers.

This built-in Initial Context Factory Builder service must be registered with no service.ranking property. This will give it the default ranking and allows other providers to override the default.

126.6 OSGi URL Scheme

A URL scheme is available that allows JNDI based applications to access services in the service registry, see *Services and State* on page 572 about restrictions on these services. The URL scheme is specified as follows:

```
service   ::= 'osgi:service/' query
query     ::= jndi-name |  qname ( '/' filter )?
jndi-name ::= <any string>
```

No spaces are allowed between the terms.

This OSGi URL scheme can be used to perform a lookup of a single matching service using the interface name and filter. The URL Context must use the *owning bundle* to perform the service queries. The owning bundle is the bundle that requested the initial Context from the JNDI Context Manager service or received its Context through the InitialContext class. The returned objects must not be incompatible with the class space of the owning bundle.

The lookup for a URL with the osgi: scheme and service path returns the service whose ServiceReference is first in ranking order as specified in ServiceReference.compareTo. This scheme only allows a single service to be found. Multiple services can be obtained with the osgi: scheme and servicelist path:

```
servicelist ::= 'osgi:servicelist/' query?
```

If this osgi:servicelist scheme is used from a lookup method then a Context object is returned instead of a service object. Calling the listBindings method will produce a NamingEnumeration object that provides Binding objects. A Binding object contains the name, class of the service, and the service object. The bound object is the service object contained in the given Context.

When the Context class list method is called, the Naming Enumeration object provides a NameClassPair object. This NameClassPair object will include the name and class of each service in the Context. The list method can be useful in cases where a client wishes to iterate over the available services without actually getting them. If the service itself is required, then listBindings method should be used.

If multiple services matched the criteria listed in the URL, there would be more than one service available in the Context, and the corresponding Naming Enumeration would contain the same number of services.

If multiple services match, a call to listBindings on this Context would return a list of bindings whose name are a string with the service.id number, for example:

```
1283
```

Thus the following lookup is valid:

```
osgi:servicelist/javax.sql.DataSource/(&(db=mydb)(version=3.1))
```

A service can provide a *JNDI service name* if it provides the following service property:

- osgi.jndi.service.name - An alternative name that the service can be looked up by when the osgi: URL scheme is used.

If a service is published with a JNDI service name then the service matches any URL that has this service name in the place of interface. For example, if the JNDI service name is foo, then the following URL selects this service:

```
osgi:service/foo
```

Using a JNDI service name that can be interpreted as an interface name must be avoided, if this happens the result is undefined.

A JNDI client can also obtain the Bundle Context of the owning bundle by using the osgi: scheme namespace with the framework/bundleContext name. The following URL must return the Bundle Context of the owning bundle:

```
osgi:framework/bundleContext
```

After the NamingEnumeration object has been used it must be closed by the client. Implementations must then unget any gotten services or perform other cleanup.

126.6.1 Service Proxies

The OSGi URL Context handles the complexities by hiding the dynamic nature of OSGi. The OSGi URL Context must handle the dynamics by *proxying* the service objects. This proxy must implement the interface given in the URL. If the JNDI service name instead of a class name is used, then all interfaces under which the service is registered must be implemented. If an interface is not compatible with the owning bundle's class space then it must not be implemented on the proxy, it must then be ignored. If this results in no implemented interfaces then an Illegal Argument Exception must be thrown.

Interfaces can always be proxied but classes are much harder. For this reason, an implementation is free to throw an Illegal Argument Exception when a class is used in the URL or in one of the registration names.

Getting the actual service object can be delayed until the proxy is actually used to call a method. If a method is called and the actual service has been unregistered, then the OSGi URL Context must attempt to rebind it to another service that matches the criteria given in the URL the next time it is called. When no alternative service is available, a Service Exception with the UNREGISTERED type code must be thrown. Services obtained with the osgi: URL scheme must therefore be stateless because the rebinding to alternative services is not visible to the caller; there are no listeners defined for this rebinding, see *Services and State* on page 572.

If the reference was looked up using osgi:servicelist then proxies must still be used, however, these proxies must not rebind when their underlying service is unregistered. Instead, they must throw a Service Exception with the UNREGISTERED type whenever the proxy is used and the proxied service is no longer available.

126.6.2 Services and State

A service obtained through a URL Context lookup is proxied. During the usage of this service, the JNDI Implementation can be forced to transparently rebind this service to another instance. The JNDI specification is largely intended for portability. For this reason, it has no mechanism architected to receive notifications about this rebinding. The client code is therefore unable to handle the dynamics.

The consequence of this model is that stateful services require extra care because applications cannot rely on the fact that they always communicate with the same service. Virtually all OSGi specified services have state.

126.7 Traditional Client Model

A JNDI Implementation must at startup register the InitialContextFactoryBuilder object and the ObjectFactoryBuilder object with the NamingManager class. As described in *JNDI Overview* on page 562, the JNDI code in the JRE will then delegate all Context related requests to the JNDI Implementation. Setting these singletons allows code that is not aware of the OSGi framework to use Context implementations from JNDI Providers registered with the OSGi service registry and that are managed as bundles. The JNDI Implementation therefore acts as a broker to the service registry for OSGi unaware code.

This brokering role can only be played when the JNDI Implementation can set the singletons as specified in *Naming Manager Singletons* on page 563. If the JNDI Implementation cannot set these singletons then it should log an error with the Log Service, if available. It can then not perform the following sections.

126.7.1 New Initial Context

The client typically requests a Context using the following code:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
InitialContext ctx = new InitialContext(env);
```

The created InitialContext object is a facade for the real Context that is requested by the caller. It provides the bootstrapping mechanism for JNDI Provider plugability. In order to obtain the provider's Context, the InitialContext class makes a call to the static getContext method on the NamingManager class. The JNDI code in the JRE then delegates any request for an initial Context object to the JNDI Implementation through the registered InitialContextFactoryBuilder singleton. The JNDI Implementation then determines the Bundle Context of the caller as described in *Caller's Bundle Context* on page 573. If no such Bundle Context can be found, a No Initial Context Exception is thrown to the caller. This Bundle Context must be from an ACTIVE bundle.

This Bundle Context is then used to get the JNDI Context Manager service. This service is then used as described in *Context Creation* on page 565 to get an initial Context. This initial Context is then used in the InitialContext object as the *default initial context*. In this specification this is normally called the backing context. An InitialContext object constructed through an Initial Context Factory Builder will not use the URL lookup mechanism, it must delegate all operations to the its backing context. A Context obtained through the JNDI Context Manager provides the URL lookup behavior instead.

126.7.2 Static Conversion

JNDI provides a general object conversion facility that is used by the URL Context and the process of restoring an object from a Reference object, see *Object and Reference Conversion* on page 563. A JNDI Implementation must take over this conversion by setting the static Object Factory Builder singleton, see *Naming Manager Singletons* on page 563. Non-OSGi aware Context implementations will use the NamingManager static getObjectInstance method for object conversion. This method then delegates to the set singleton Object Factory Builder to obtain an ObjectFactory object that understands how to convert the given description to an object. The JNDI Implementation must return an Object Factory that understands the OSGi service registry. If the getObjectInstance method is called on this object it must use the same rules as defined for the JNDI Provider Admin service getObjectInstance(Object,javax.naming.Name,javax.naming.Context,Map) method, see *JNDI Provider Admin service* on page 566. The Bundle Context that must be used with respect to this service is the caller's Bundle Context, see *Caller's Bundle Context* on page 573. If the Bundle Context is not found, the description object must be returned. The calling bundle must not be required to import the org.osgi.service.jndi package.

126.7.3 Caller's Bundle Context

The following mechanisms are used to determine the callers Bundle Context:

1. Look in the JNDI environment properties for a property called

   ```
   osgi.service.jndi.bundleContext
   ```

   If a value for this property exists then use it as the Bundle Context. If the Bundle Context has been found stop.
2. Obtain the Thread Context Class Loader; if it, or an ancestor class loader, implements the BundleReference interface, call its getBundle method to get the client's Bundle; then call getBundleContext on the Bundle object to get the client's Bundle Context. If the Bundle Context has been found stop.
3. Walk the call stack until the invoker is found. The invoker can be the caller of the InitialContext class constructor or the NamingManager or DirectoryManager getObjectInstance methods.
   - Get the class loader of the caller and see if it, or an ancestor, implements the BundleReference interface.
   - If a Class Loader implementing the BundleReference interface is found call the getBundle method to get the clients Bundle; then call the getBundleContext method on the Bundle to get the clients Bundle Context.
   - If the Bundle Context has been found stop, else continue with the next stack frame.

126.7.4 Life Cycle Mismatch

The use of static access to the JNDI mechanisms, NamingManager and InitialContext class methods, in the traditional client programming model produces several problems with regard to the OSGi life cycle. The primary problem being that there is no dependency management in place when static methods are used. These problems do not exist for the JNDI Context Manager service. Therefore, OSGi applications are strongly encouraged to use the JNDI Context Manager service.

The traditional programming model approach relies on two JVM singletons in the Naming Manager, see *Naming Manager Singletons* on page 563. The JNDI Implementation bundle must set both singletons before it registers its JNDI Context Manager service and JNDI Provider Admin service. However, in OSGi there is no defined start ordering, primarily because bundles can be updated at any moment in time and will at such time not be available to provide their function anyway. For this reason, OSGi bundles express their dependencies with services.

The lack of start ordering means that a bundle could create an InitialContext object before the JNDI Implementation has had the chance to set the static Initial Context Factory Builder singleton. This means that the JNDI implementation inside the JRE will provide its default behavior and likely have to throw an exception. A similar exception is thrown for the Object Factory Builder singleton.

There is a also a (small) possibility that a client will call new InitialContext() after the singletons have been set, but before the JNDI Context Manager and JNDI Provider Admin services have been registered. This specification requires that these services are set after the singletons are set. In this race condition the JNDI Implementation should throw a No Initial Context Exception, explaining that the JNDI services are not available yet.

126.8 Security

126.8.1 JNDI Implementation

A JNDI Implementation may wish to assert that the user of the provider has some relevant Java 2 security permission. Since the JNDI implementation is an intermediary between the JNDI client and provider this means that the JNDI implementation needs to have any permissions required to access any JNDI Provider. As a result the JNDI implementation needs All Permission. This will result in the JNDI clients permissions being checked to see if it has the relevant permission to access the JNDI Provider.

The JNDI Implementation must make any invocation to access these services in a doPriviledged check. A JNDI client must therefore not be required to have the following permissions, which are needed by a JNDI Implementation:

```
ServicePermission ..ObjectFactory                REGISTER,GET
ServicePermission ..DirObjectFactory             REGISTER,GET
ServicePermission ..ObjectFactoryBuilder         REGISTER,GET
ServicePermission ..InitialContextFactory        REGISTER,GET
ServicePermission ..InitialContextFactoryBuilder REGISTER,GET
ServicePermission ..JNDIProviderAdmin            REGISTER,GET
```

The JNDI Implementation bundle must have the appropriate permissions to install the InitialContextFactoryBuilder and ObjectFactoryBuilder instances using the appropriate methods on the NamingManager class. This requires the following permission:

```
RuntimePermission "setFactory"
```

126.8.2 JNDI Clients

A JNDI client using the JNDI Context Manager service must have the following permissions:

```
ServicePermission ..JNDIContextManager           GET
```

Obtaining a reference to a JNDI Context Manager service should be considered a privileged operation and should be guarded by permissions.

126.8.3 OSGi URL namespace

A JNDI client must not be able to obtain services or a Bundle Context that the client bundle would not be able to get via the core OSGi API. To allow a client to use the osgi namespace to get a service the bundle must have the corresponding Service Permission. When using the osgi namespace to obtain the Bundle Context the client bundle must have Admin Permission for the Bundle Context.

These permissions must be enforced by the osgi URL namespace handler. If there is no proper permission, the implementation must throw a Name Not Found Exception to prevent exposing the existence of such services.

126.9 org.osgi.service.jndi

JNDI Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jndi; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jndi; version="[1.0,1.1)"
```

126.9.1 Summary

- `JNDIConstants` - Constants for the JNDI implementation.
- `JNDIContextManager` - This interface defines the OSGi service interface for the JNDIContextManager.
- `JNDIProviderAdmin` - This interface defines the OSGi service interface for the JNDIProviderAdmin service.

126.9.2 public class JNDIConstants

Constants for the JNDI implementation.

Concurrency Immutable

126.9.2.1 public static final String BUNDLE_CONTEXT = "osgi.service.jndi.bundleContext"

This JNDI environment property can be used by a JNDI client to indicate the caller's BundleContext. This property can be set and passed to an InitialContext constructor. This property is only useful in the "traditional" mode of JNDI.

126.9.2.2 public static final String JNDI_SERVICENAME = "osgi.jndi.service.name"

This service property is set on an OSGi service to provide a name that can be used to locate the service other than the service interface name.

126.9.2.3 public static final String JNDI_URLSCHEME = "osgi.jndi.url.scheme"

This service property is set by JNDI Providers that publish URL Context Factories as OSGi Services. The value of this property should be the URL scheme that is supported by the published service.

126.9.3 public interface JNDIContextManager

This interface defines the OSGi service interface for the JNDIContextManager. This service provides the ability to create new JNDI Context instances without relying on the InitialContext constructor.

Concurrency Thread-safe

126.9.3.1 public Context newInitialContext() throws NamingException

□ Creates a new JNDI initial context with the default JNDI environment properties.

Returns an instance of javax.naming.Context

Throws NamingException– upon any error that occurs during context creation

126.9.3.2 public Context newInitialContext(Map<String, ?> environment) throws NamingException

environment JNDI environment properties specified by caller

□ Creates a new JNDI initial context with the specified JNDI environment properties.

Returns an instance of javax.naming.Context

Throws NamingException– upon any error that occurs during context creation

126.9.3.3 public DirContext newInitialDirContext() throws NamingException

□ Creates a new initial DirContext with the default JNDI environment properties.

Returns an instance of javax.naming.directory.DirContext

Throws NamingException– upon any error that occurs during context creation

126.9.3.4 public DirContext newInitialDirContext(Map<String, ?> environment) throws NamingException

environment JNDI environment properties specified by the caller

□ Creates a new initial DirContext with the specified JNDI environment properties.

Returns an instance of javax.naming.directory.DirContext

Throws NamingException– upon any error that occurs during context creation

126.9.4 public interface JNDIProviderAdmin

This interface defines the OSGi service interface for the JNDIProviderAdmin service. This service provides the ability to resolve JNDI References in a dynamic fashion that does not require calls to NamingManager.getObjectInstance(). The methods of this service provide similar reference resolution, but rely on the OSGi Service Registry in order to find ObjectFactory instances that can convert a Reference to an Object. This service will typically be used by OSGi-aware JNDI Service Providers.

Concurrency Thread-safe

126.9.4.1 public Object getObjectInstance(Object refInfo, Name name, Context context, Map<String, ?> environment) throws Exception

refInfo Reference info

name the JNDI name associated with this reference

context the JNDI context associated with this reference

environment the JNDI environment associated with this JNDI context

□ Resolve the object from the given reference.

Returns an Object based on the reference passed in, or the original reference object if the reference could not be resolved.

Throws Exception– in the event that an error occurs while attempting to resolve the JNDI reference.

126.9.4.2 public Object getObjectInstance(Object refInfo, Name name, Context context, Map<String, ?> environment, Attributes attributes) throws Exception

refInfo Reference info

name the JNDI name associated with this reference

context the JNDI context associated with this reference

environment the JNDI environment associated with this JNDI context *attributes* the naming attributes to use when resolving this object

□ Resolve the object from the given reference.

Returns an Object based on the reference passed in, or the original reference object if the reference could not be resolved.

Throws `Exception`– in the event that an error occurs while attempting to resolve the JNDI reference.

126.10 References

[1] *Java Naming and Directory Interface*
https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html

[2] *Java Naming and Directory Interface Tutorial*
https://docs.oracle.com/javase/tutorial/jndi/index.html

[3] *JNDI Standard Property Names*
https://docs.oracle.com/javase/8/docs/api/javax/naming/Context.html

127 JPA Service Specification

Version 1.1

127.1 Introduction

The Java Persistence API (JPA) is a specification that sets a standard for persistently storing objects in enterprise and non-enterprise Java based environments. JPA provides an Object Relational Mapping (ORM) model that is configured through persistence descriptors. This Java Persistence Service specification defines how persistence units can be published in an OSGi framework, how client bundles can find these persistence units, how database drivers are found with the *Data Service Specification for JDBC™ Technology* on page 551, as well as how JPA providers can be made available within an OSGi framework.

Applications can be managed or they can be unmanaged. Managed applications run inside a Java EE Container and unmanaged applications run in a Java SE environment. The managed case requires a provider interface that can be used by the container, while in the unmanaged case the JPA provider is responsible for supporting the client directly. This specification is about the unmanaged model of JPA except in the areas where the managed model is explicitly mentioned. Additionally, multiple concurrent providers for the unmanaged case are not supported.

127.1.1 Essentials

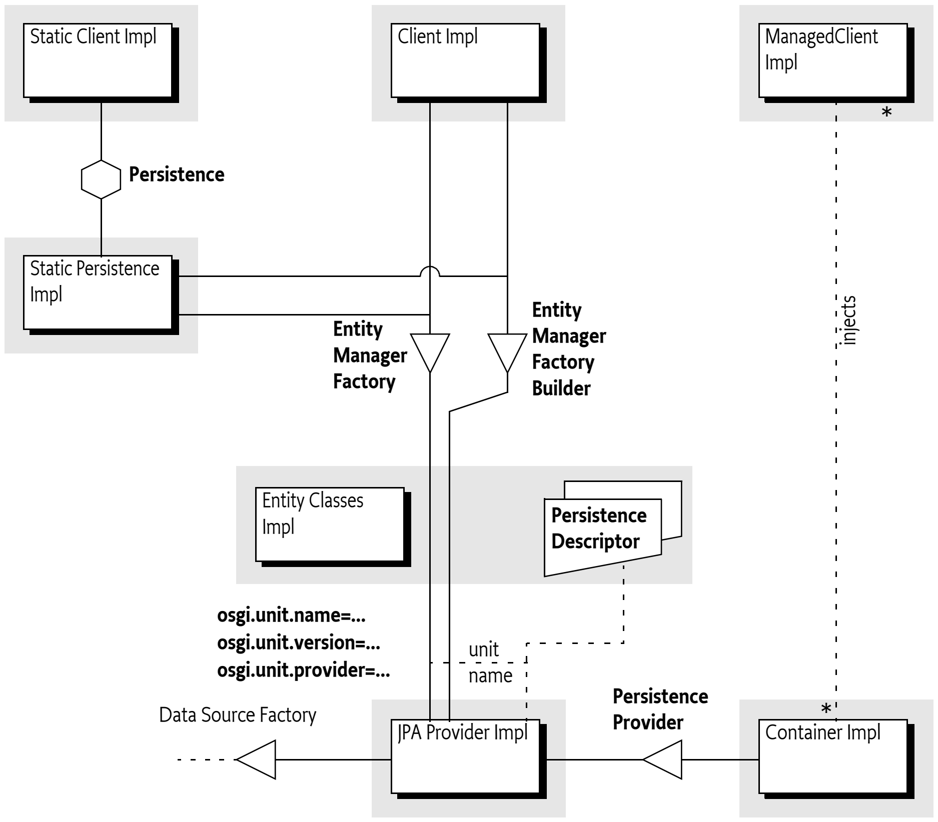
- *Dependencies* - There must be a way for persistence clients, if they so require, to manage their dependencies on a compatible persistence unit.
- *Compatibility* - The Persistence Unit service must be able to function in non-managed mode according to existing standards and interfaces outlined in the JPA specification.
- *Modularity* - Persistent classes and their accompanying configuration can exist in a separate bundle from the client that is operating on them using the Persistence Unit service.
- *JDBC* - Leverage the *Data Service Specification for JDBC™ Technology* on page 551 for access to the database.

127.1.2 Entities

- *JPA* - The Java Persistence API, [3] *JPA 2.1*.
- *JPA Provider* - An implementation of JPA, providing the Persistence Provider and JPA Services to Java EE Containers and Client Bundles.
- *Interface Bundle* - A bundle containing the interfaces and classes in the `javax.persistence` namespace (and its sub-namespaces) that are defined by the JPA specification.
- *Persistence Bundle* - A bundle that includes, a Meta-Persistence header, one or more Persistence Descriptor resources, and the entity classes specified by the Persistence Units in those resources.
- *Client Bundle* - The bundle that uses the Persistence Bundle to retrieve and store objects.
- *Persistence Descriptor* - A resource describing one or more Persistence Units.
- *Persistence Unit* - A named configuration for the object-relational mappings and database access as defined in a Persistence Descriptor.
- *Entity Manager* - The interface that provides the control point of retrieving and persisting objects in a relational database based on a single Persistence Unit for a single session.

- *Entity Manager Factory* - A service that can create Entity Managers based on a Persistence Unit for different sessions.
- *Entity Manager Factory Builder* - A service that can build an Entity Manager Factory for a specific Persistence Unit with extra configuration parameters.
- *Managed Client* - A Client Bundle that is managed by a Container
- *Static Client* - A Client that uses the static factory methods in the Persistence class instead of services.
- *Static Persistence* - The actor that enables the use of the `Persistence` class static factory methods to obtain an Entity Manager Factory.
- *JDBC Provider* - The bundle providing a Data Source Factory service.

Figure 127.1 *JPA Service overview*



127.1.3 Dependencies

This specification requires a minimum JPA version of 2.1. Implementations may choose to support newer versions of JPA, for example version 2.2, but must offer the JavaJPA contract at version 2.1 as well as any future versions that they support.

127.1.4 Synopsis

A JPA Provider tracks Persistence Bundles; a Persistence Bundle contains a Meta-Persistence manifest header. This manifest header enumerates the Persistence Descriptor resources in the Persistence Bundle. Each resource's XML schema is defined by the JPA specification. The JPA Provider reads the resource accordingly and extracts the information for one or more Persistence Units. For each found Persistence Unit, the JPA Provider registers an Entity Manager Factory Builder service. If the database is defined in the Persistence Unit, then the JPA Provider registers an Entity Manager Factory service during the availability of the corresponding Data Source Factory.

The identification of these services is handled through a number of service properties. The Entity Manager Factory service is named by the standard JPA interface, the Builder version is OSGi specific; it is used when the Client Bundle needs to create an Entity Manager Factory based on configuration properties.

A Client Bundle that wants to persist or retrieve its entity classes depends on an Entity Manager Factory (Builder) service that corresponds to a Persistence Unit that lists the entity classes. If such a service is available, the client can use this service to get an Entity Manager, allowing the client to retrieve and persist objects as long as the originating Entity Manager Factory (Builder) service is registered.

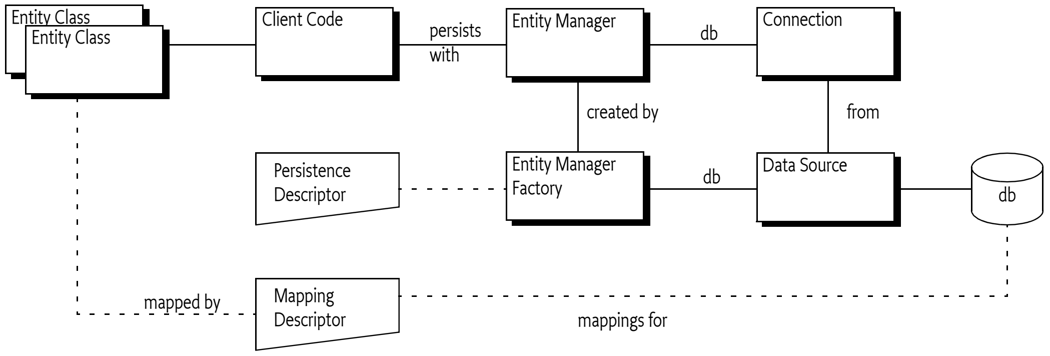
In a non-OSGi environment, it is customary to get an Entity Manager Factory through the `Persistence` class. This `Persistence` class provides a number of static methods that give access to any locally available JPA providers. This approach is not recommended in an OSGi environment due to class loading and start ordering issues. However, OSGi environments can support access through this static factory with a Static Persistence bundle.

127.2 JPA Overview

Java Persistence API (JPA) is a specification that is part of [4] *Java EE 5*. This OSGi Specification is based on [1] *JPA 1.0*, [2] *JPA 2.0* and [3] *JPA 2.1*. This section provides an overview of JPA as specified in the JCP. The purpose of this section is to introduce the concepts behind JPA and define the terminology that will be used in the remainder of the chapter.

The purpose of JPA is to simplify access to relational databases for applications on the object-oriented Java platform. JPA provides support for storing and retrieving objects in a relational database. The JPA specification defines in detail how objects are mapped to tables and columns under the full control of the application. The core classes involved are depicted in Figure 127.2.

Figure 127.2 JPA Client View



The JPA specifications define a number of concepts that are defined in this section for the purpose of this OSGi specification. However, the full syntax and semantics are defined in the JPA specifications.

127.2.1 Persistence

Classes that are stored and retrieved through JPA are called the *entity classes*. In this specification, the concept of entity classes includes the *embeddable* classes, which are classes that do not have any persistent identity, and mapped super classes that allow mappings, but are not themselves persistent. Entity classes are not required to implement any interface or extend a specific superclass, they are Plain Old Java Objects (POJOs). It is the responsibility of the *JPA Provider* to connect to a database and map the store and retrieve operations of the entity classes to their tables and columns. For performance reasons, the entity classes are sometimes *enhanced*. This enhancement can take place during build time, deploy time, or during class loading time. Some enhancements use byte code weaving, some enhancements are based on sub-classing.

The JPA Provider cannot automatically perform its persistence tasks; it requires configuration information. This configuration information is stored in the *Persistence Descriptor*. A Persistence Descriptor is an XML file according of one of the two following namespaces:

```
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
```

The JPA standard Persistence Descriptor must be stored in `META-INF/persistence.xml`. It is usually in the same class path entry (like a JAR or directory) as the entity classes.

The JPA Provider parses the Persistence Descriptor and extracts one or more *Persistence Units*. A Persistence Unit includes the following aspects:

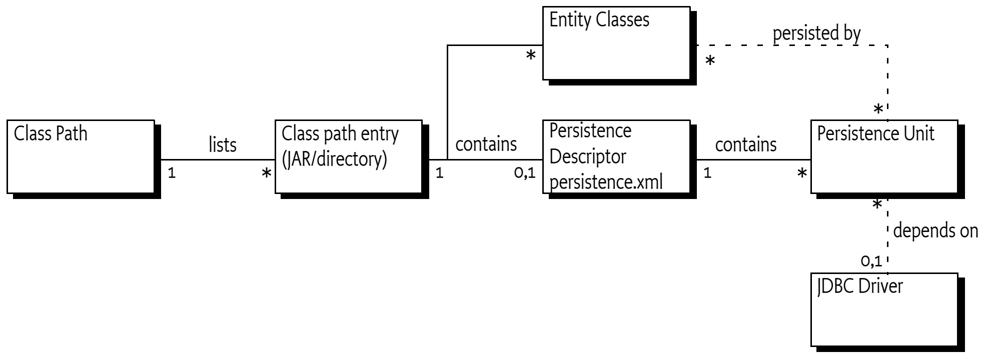
- *Name* - Every Persistence Unit must have a name to identify it to clients. For example: `Accounting`.
- *Provider Selection* - Restriction to a specific JPA Provider, usually because there are dependencies in the application code on provider specific functionality.
- *JDBC Driver Selection* - Selects the JDBC driver, the principal and the credentials for selecting and accessing a relational database. See *JDBC Access in JPA* on page 584.
- *Properties* - Standard and JPA Provider specific properties.

The object-relational mappings are stored in special mapping resources or are specified in annotations.

A Persistence Unit can be *complete* or *incomplete*. A complete Persistence Unit identifies the database driver that is needed for the Persistence Unit, though it does not have to contain the credentials. An incomplete Persistence Unit lacks this information.

The relations between the class path, its entries, the entity classes, the Persistence Descriptor and the Persistence Unit is depicted in Figure 127.3 on page 582.

Figure 127.3 *JPA Configuration*



JPA recognizes the concept of a *persistence root*. The persistence root is the root of the JAR (or directory) on the class path that contains the `META-INF/persistence.xml` resource.

127.2.2 JPA Provider

The JPA specifications provide support for multiple JPA Providers in the same application. An Application selects a JPA Provider through the `Persistence` class, using static factory methods. One of these methods accepts a map with *configuration properties*. Configuration properties can override information specified in a Persistence Unit or these properties add new information to the Persistence Unit.

The default implementation of the Persistence class discovers providers through the Java EE services model, this model requires a text resource in the class path entry called:

```
META-INF/services/javax.persistence.PersistenceProvider
```

This text resource contains the name of the JPA Provider implementation class.

The Persistence class createEntityManagerFactory method provides the JPA Provider with the name of a Persistence Unit. The JPA Provider must then scan the class path for any META-INF/persistence.xml entries, these are the available Persistence Descriptors. It then extracts the Persistence Units to find the requested Persistence Unit. If no such Persistence Unit can be found, or the JPA Provider is restricted from servicing this Persistence Unit, then null is returned. The Persistence class will then continue to try the next found or registered JPA Provider.

A Persistence Unit can restrict JPA Providers by specifying a *JPA Provider class*, this introduces a *provider dependency*. The specified JPA Provider class must implement the PersistenceProvider interface. This *implementation class name* must be available from the JPA Provider's documentation. JPA Providers that do not own the specified JPA Provider class must ignore such a Persistence Unit.

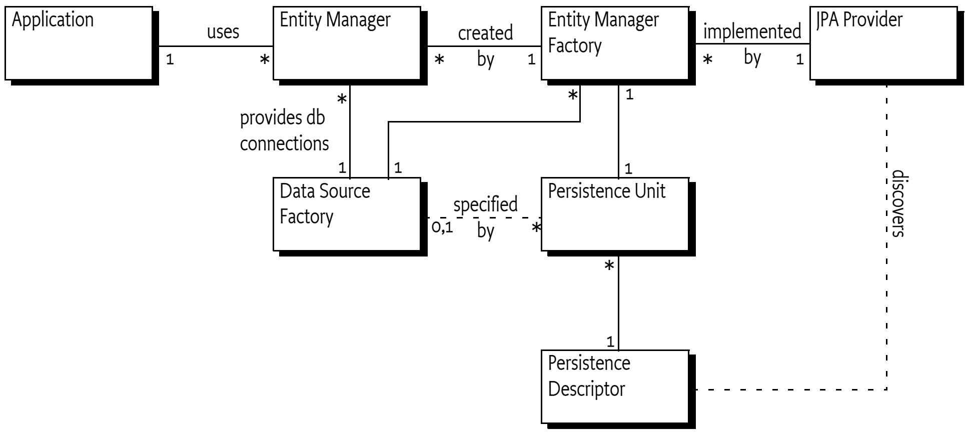
Otherwise, if the Persistence Unit is not restricted, the JPA Provider is *assigned* to this Persistence Unit; it must be ready to provide an EntityManagerFactory object when the application requests one.

The JPA Provider uses the Persistence Unit, together with any additional configuration properties, to construct an *Entity Manager Factory*. The application then uses this Entity Manager Factory to construct an *Entity Manager*, optionally providing additional configuration properties. The Entity Manager then provides the operations for the application to store and retrieve entity classes from the database.

The additional configuration properties provided with the creation of the Entity Manager Factory or the Entity Manager are often used to specify the database driver and the credentials. This allows the Persistence Unit to be specified without committing to a specific database, leaving the choice to the application at runtime.

The relations between the application, Entity Manager, Entity Manager Factory and the JPA Provider are depicted in Figure 127.4 on page 583.

Figure 127.4 *JPA Dynamic Model*



127.2.3 Managed and Unmanaged

The JPA specifications make a distinction between a *managed* and an *unmanaged* mode. In the managed mode the presence of a Java EE Container is assumed. Such a container provides many services for its contained applications like transaction handling, dependency injection, etc. One of these aspects can be the interface to the relational database. The JPA specifications therefore have defined a special method for Java EE Containers to manage the persistence aspects of their Managed Clients. This method is the createContainerEntityManagerFactory method on the PersistenceProvider interface. This method is purely intended for Java EE Containers and should not be used in other environments.

The other method on the PersistenceProvider interface is intended to be used by the Persistence class static factory methods. The Persistence class searches for an appropriate JPA Provider by asking all available JPA Providers to create an Entity Manager Factory based on configuration properties. The first JPA Provider that is capable of providing an Entity Manager Factory wins. The use of these static factory methods is called the *unmanaged mode*. It requires a JPA Provider to scan the class path to find the assigned Persistence Units.

127.2.4 JDBC Access in JPA

A Persistence Unit is configured to work with a relational database. JPA Providers communicate with a relational database through compliant JDBC database drivers. The database and driver parameters are specified in the Persistence Unit or configured during Entity Manager Factory or Entity Manager creation with the configuration properties. The configuration properties for selecting a database in non-managed mode were proprietary in JPA 1.0 but have been standardized in version 2.0 of JPA:

- javax.persistence.jdbc.driver - Fully-qualified name of the driver class
- javax.persistence.jdbc.url - Driver-specific URL to indicate database information
- javax.persistence.jdbc.user - User name to use when obtaining connections
- javax.persistence.jdbc.password - Password to use when obtaining connections

127.3 Bundles with Persistence

The primary goal of this specification is to simplify the programming model for bundles that need persistence. In this specification there are two application roles:

- *Persistence Bundle* - A Persistence Bundle contains the entity classes and one or more Persistence Descriptors, each providing one or more Persistence Units.
- *Client Bundle* -A Client Bundle contains the code that manipulates the entity classes and uses an Entity Manager to store and retrieve these entity classes with a relational database. The Client Bundle obtains the required Entity Manager(s) via a service based model.

These roles can be combined in a single bundle.

127.3.1 Services

A JPA Provider uses Persistence Units to provide Client Bundles with a configured *Entity Manager Factory* service and/or an *Entity Manager Factory Builder* service for each assigned Persistence Unit:

- *Entity Manager Factory service* - Provides an EntityManagerFactory object that depends on a complete Persistence Unit. That is, it is associated with a registered Data Source Factory service.
- *Entity Manager Factory Builder service* - The Entity Manager Factory Builder service provides the capability of creating an EntityManagerFactory object with additional configuration properties. The Entity Manager Factory Builder service also provides information about the JPA Provider that will be used to create the EntityManagerFactory object.

These services are collectively called the *JPA Services*. Entity Managers obtained from such JPA Services can only be used to operate on entity classes associated with their corresponding Persistence Unit.

127.3.2 Persistence Bundle

A *Persistence Bundle* is a bundle that specifies the Meta-Persistence header, see *Meta Persistence Header* on page 587. This header refers to one or more Persistence Descriptors in the Persistence Bundle. Commonly, this is the META-INF/persistence.xml resource. This location is the standard for non-OSGi environments, however an OSGi bundle can also use other locations as well as multiple resources.

For example, the contents of a simple Persistence Bundle with a single Person entity class could look like:

```
META-INF/
META-INF/MANIFEST.MF
OSGI-INF/address.xml
com/acme/Person.class
```

The corresponding manifest would then look like:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Meta-Persistence: OSGI-INF/address.xml
Bundle-SymbolicName: com.acme.simple.persistence
Bundle-Version: 3.2.4.200912231004
```

A Persistence Bundle is a normal bundle; it must follow all the rules of OSGi and can use all OSGi constructs like Bundle-ClassPath, fragment bundles, import packages, export packages, etc. However, there is one limitation: any entity classes must originate in the bundle's JAR, it cannot come from a fragment. This requirement is necessary to simplify enhancing entity classes.

127.3.3 Client Bundles

A Client Bundle uses the entity classes from a Persistence Bundle to provide its required functionality. To store and retrieve these entity classes a Client Bundle requires an Entity Manager that is configured for the corresponding Persistence Unit.

An Entity Manager is intended to be used by a single session, it is not thread safe. Therefore, a client needs an Entity Manager Factory to create an Entity Manager. In an OSGi environment, there are multiple routes to obtain an Entity Manager Factory.

A JPA Provider must register an Entity Manager Factory service for each assigned Persistence Unit that is *complete*. Complete means that it is a configured Persistence Unit, including the reference to the relational database. The Entity Manager Factory service is therefore bound to a Data Source Factory service and Client Bundles should not attempt to rebind the Data Source Factory with the configuration properties of the createEntityManager(Map) method. See *Rebinding* on page 592 for the consequences. If the Data Source Factory must be bound by the Client Bundle then the Client Bundle should use the *Custom Configured Entity Manager* on page 586.

The Entity Manager Factory service must be registered with the service properties as defined in *Service Registrations* on page 589. These are:

- osgi.unit.name - (String) The name of the Persistence Unit
- osgi.unit.version - (String) The version of the associated Persistence Bundle
- osgi.unit.provider - (String) The implementation class name of the JPA Provider

The life cycle of the Entity Manager Factory service is bound to the Persistence Bundle, the JPA Provider, and the selected Data Source Factory service.

A Client Bundle that wants to use an Entity Manager Factory service should therefore use an appropriate filter to select the Entity Manager Factory service that corresponds to its required Persistence

Unit. For example, the following snippet uses Declarative Services, see *Declarative Services Specification* on page 245, to statically depend on such a service:

```
<reference name="accounting"
   target="(&amp;(osgi.unit.name=Accounting)(osgi.unit.version=3.2.*))"
    interface="javax.persistence.EntityManagerFactory"/>
```

127.3.4 Custom Configured Entity Manager

If a Client Bundle needs to provide configuration properties for the creation of an Entity Manager Factory it should use the *Entity Manager Factory Builder* service. This can for example be used to provide the database selection properties when the Persistence Unit is incomplete or if the database selection needs to be overridden. The Entity Manager Factory Builder service also provides information about the JPA Provider that will be used to create the Entity Manager Factory. This information can be used by the Client Bundle when determining what (if any) JPA Provider implementation specific configuration that the Client Bundle will provide.

The Entity Manager Factory Builder service's life cycle must not depend on the availability of any Data Source Factory, even if a JDBC driver class name is specified in the Persistence Descriptor. The Entity Manager Factory Builder service is registered with the same service properties as the corresponding Entity Factory service, see *Service Registrations* on page 589.

The following methods are defined on the EntityManagerFactoryBuilder interface:

- createEntityManagerFactory(Map) - Returns a custom configured EntityManagerFactory instance for the Persistence Unit associated with the service. Accepts a map with the configuration properties to be applied during Entity Manager Factory creation. The method must return a proper Entity Manager Factory or throw an Exception.
- getPersistenceProviderName() - Returns the name of the PersistenceProvider implementation class used in Entity Manager Factory creation. This name will be the same as the value of the JPA_UNIT_PROVIDER service property.
- getPersistenceProviderBundle() - Returns the bundle JPA Provider implementation bundle which provides the PersistenceProvider. If the Persistence Provider was provided as an OSGi service then this method must return the bundle which registered the service. Otherwise this method must return the bundle which loaded the PersistenceProvider implementation class.

The createEntityManagerFactory method allows standard and vendor-specific properties to be passed in and applied to the Entity Manager Factory being created. However, some properties cannot be honored by the aforementioned method. For example, the javax.persistence.provider JPA property, as a means to specify a specific JPA Provider at runtime, cannot be supported because the JPA Provider has already been decided; it is the JPA Provider that registered the Entity Manager Factory Builder service. A JPA Provider should throw an Exception if it recognizes the property but it cannot use the property when specified through the builder. Unrecognized properties must be ignored.

Once an Entity Manager Factory is created the specified Data Source becomes associated with the Entity Manager Factory. It is therefore not possible to re-associate an Entity Manager Factory with another Data Source by providing different properties. A JPA Provider must throw an Exception when an attempt is made to re-specify the database properties. See *Rebinding* on page 592 for further information.

As an example, a sample snippet of a client that wants to operate on a persistence unit named Accounting and pass in the JDBC user name and password properties is:

```
ServiceReference[] refs = context.getServiceReferences(
    EntityManagerFactoryBuilder.class.getName(),
    "(osgi.unit.name=Accounting)");
if ( refs != null ) {
    EntityManagerFactoryBuilder emfBuilder =
        (EntityManagerFactoryBuilder) context.getService(refs[0]);
    if ( emfBuilder != null ) {
      Map<String,Object> props = new HashMap<String,Object>();
      props.put("javax.persistence.jdbc.user", userString);
      props.put("javax.persistence.jdbc.password",passwordString);
      EntityManagerFactory emf = emfBuilder.createEntityManagerFactory(props);
      EntityManager em = emf.createEntityManager();
     ...
}
```

The example does not handle the dynamic dependencies on the associated Data Source Factory service.

127.3.4.1 Supported configuration properties

The [3] *JPA 2.1* specification adds a significant number of standard property names. These properties are used both for runtime control, and also for configuring JPA persistence units as they are created.

The EntityManagerFactoryBuilder service must support the defined property names as per the JPA specification. In most cases this will be accomplished by passing the values directly to the Persistence Provider, but in some cases it may require further action from the JPA Service implementation.

127.4 Extending a Persistence Bundle

A Persistence Bundle is identified by its Meta-Persistence manifest header that references a number of Persistence Descriptor resources. Persistence bundles must be detected by a JPA Provider. The JPA Provider must parse any Persistence Descriptors in these bundles and detect the assigned Persistence Units. For each assigned Persistence Unit, the JPA Provider must register an Entity Manager Factory Builder service when the Persistence Bundle is ready, see *Ready Phase* on page 589.

For complete and assigned Persistence Units, the JPA Provider must find the required Data Source Factory service based on the driver name. When the Persistence Bundle is ready and the selected Data Source Factory is available, the JPA Provider must have an Entity Manager Factory service registered that is linked to that Data Source Factory.

When the Persistence Bundle is stopped (or the JPA Provider stops), the JPA Provider must close all connections and cleanup any resources associated with the Persistence Bundle.

This process is outlined in detail in the following sections.

127.4.1 Class Space Consistency

A JPA Provider must ignore Persistence Bundles that are in another class space for the javax.persistence.* packages. Such a JPA Provider cannot create JPA Services that would be visible and usable by the Client Bundles.

127.4.2 Meta Persistence Header

A *Persistence Bundle* is a bundle that contains the Meta-Persistence header. If this header is not present, then this specification does not apply and a JPA Provider should ignore the corresponding bundle.

The persistence root of a Persistence Unit is the root of the Persistence Bundle's JAR

The Meta-Persistence header has a syntax of:

```
Meta-Persistence ::= ( jar-path ( ',' jar-path)* )?
jar-path         ::= path ( '!/' spath )?
```

```
spath           ::= path   // must not start with solidus ('/' \u002F)
```

The header may include zero or more comma-separated jar-paths, each a path to a Persistence Descriptor resource in the bundle. Paths may optionally be prefixed with the solidus ('/' \u002F) character. The JPA Provider must always include the META-INF/persistence.xml first if it is not one of the listed paths. Wildcards in directories are not supported. The META-INF/persistence.xml is therefore the default location for an empty header.

For example:

```
Meta-Persistence: META-INF/jpa.xml, persistence/jpa.xml
```

The previous example will instruct the JPA Provider to process the META-INF/persistence.xml resource first, even though it is not explicitly listed. The JPA Provider must then subsequently process META-INF/jpa.xml and the persistence/jpa.xml resources.

The paths in the Meta-Persistence header must be used with the Bundle.getEntry() method, or a mechanism with similar semantics, to obtain the corresponding resource. The getEntry method does not force the bundle to resolve when still unresolved; resolving might interfere with the efficiency of any required entity class enhancements. However, the use of the getEntry method implies that fragment bundles cannot be used to contain Persistence Descriptors nor entity classes.

Paths in the Meta-Persistence header can reference JAR files that are nested in the bundle by using the !/ jar: URL syntax to separate the JAR file from the path within the JAR, for example:

```
Meta-Persistence: embedded.jar!/META-INF/persistence.xml
```

This example refers to a resource in the embedded.jar resource, located in the META-INF directory of embedded.jar.

The !/ splits the jar-path in a prefix and a suffix:

- *Prefix* - The prefix is a path to a JAR resource in the bundle.
- *Suffix* - The suffix is a path to a resource in the JAR identified by the prefix.

For example:

```
embedded.jar!/META-INF/persistence.xml
prefix:     embedded.jar
suffix:     META-INF/persistence.xml
```

It is not required that all listed or implied resources are present in the bundle's JAR. For example, it is valid that the default META-INF/persistence.xml resource is absent. However, if no Persistence Units are found at all then the absence of any Persistence Unit is regarded as an error that should be logged. In this case, the Persistence Bundle is further ignored.

127.4.3 Processing

A JPA Provider can detect a Persistence Bundle as early as its installation time. This early detection allows the JPA Provider to validate the Persistence Bundle as well as prepare any mechanisms to enhance the classes for better performance. However, this process can also be delayed until the bundle is started.

The JPA Provider must validate the Persistence Bundle. A valid Persistence Bundle must:

- Have no parsing errors of the Persistence Descriptors
- Validate all Persistence Descriptors against their schemas
- Have at least one assigned Persistence Unit
- Have all entity classes mentioned in the assigned Persistence Units on the Persistence Bundle's JAR.

A Persistence Bundle that uses multiple providers for its Persistence Units could become incompatible with future versions of this specification.

If any validation fails, then this is an error and should be logged. Such a bundle is ignored completely even if it also contains valid assigned Persistence Units. Only a bundle update can recover from this state.

Persistence Units can restrict JPA Providers by specifying a provider dependency. JPA Providers that do not own this JPA Provider implementation class must ignore such a Persistence Unit completely. Otherwise, if the JPA Provider can service a Persistence Unit, it assigns itself to this Persistence Unit.

If after the processing of all Persistence Descriptors, the JPA Provider has no assigned Persistence Units, then the JPA Provider must further ignore the Persistence Bundle.

127.4.4 Ready Phase

A Persistence Bundle is *ready* when its state is `ACTIVE` or, when a `lazy` activation policy is used, `STARTING`. A JPA Provider must track the ready state of Persistence Bundles that contain assigned Persistence Units.

While a Persistence Bundle is ready, the JPA Provider must have, for each assigned Persistence Unit, an Entity Manager Factory Builder service registered to allow Client Bundles to create new `EntityManagerFactory` objects. The JPA Provider must also register an Entity Manager Factory for each assigned and complete Persistence Unit that has its corresponding Data Source available in the service registry.

The service registration process is asynchronous with the Persistence Bundle start because a JPA Provider could start after a Persistence Bundle became ready.

127.4.5 Service Registrations

The JPA Services must be registered through the Bundle Context of the corresponding Persistence Bundle to ensure proper class space consistency checks by the OSGi Framework.

JPA Services are always related to an assigned Persistence Unit. To identify this Persistence Unit and the assigned JPA Provider, each JPA Service must have the following service properties:

- `osgi.unit.name` - (`String`) The name of the Persistence Unit. This property corresponds to the `name` attribute of the `persistence-unit` element in the Persistence Descriptor. It is used by Client Bundles as the primary filter criterion to obtain a JPA Service for a required Persistence Unit. There can be multiple JPA Services registered under the same `osgi.unit.name`, each representing a different version of the Persistence Unit.
- `osgi.unit.version` - (`String`) The version of the Persistence Bundle, as specified in Bundle-Version header, that provides the corresponding Persistence Unit. Client Bundles can filter their required JPA Services based on a particular Persistence Unit version.
- `osgi.unit.provider` - (`String`) The JPA Provider implementation class name that registered the service. The `osgi.unit.provider` property allows Client Bundles to know the JPA Provider that is servicing the Persistence Unit. Client Bundles should be careful when filtering on this property, however, since the JPA Provider that is assigned a Persistence Unit may not be known by the Client Bundle ahead of time. If there is a JPA Provider dependency, it is better to specify this dependency in the Persistence Unit because other JPA Providers are then not allowed to assign such a Persistence Unit and will therefore not register a service.

127.4.6 Registering the Entity Manager Factory Builder Service

Once the Persistence Bundle is ready, a JPA Provider must register an Entity Manager Factory Builder service for each assigned Persistence Unit from that Persistence Bundle.

The Entity Manager Factory Builder service must be registered with the service properties listed in *Service Registrations* on page 589. The Entity Manager Factory Builder service is registered under the `org.osgi.service.jpa.EntityManagerFactoryBuilder` name. This interface is using the JPA packages and is therefore bound to one of the two supported versions, see *Dependencies* on page 580.

The Entity Manager Factory Builder service enables the creation of a parameterized version of an Entity Factory Manager by allowing the caller to specify configuration properties. This approach is necessary if, for example, the Persistence Unit is not complete.

127.4.7 Registering the Entity Manager Factory

A complete Persistence Unit is configured with a specific relational database driver, see *JDBC Access in JPA* on page 584. A JPA Provider must have an Entity Manager Factory service registered for each assigned and complete Persistence Unit when:

- The originating Persistence Bundle is ready, and
- A *matching* Data Source Factory service is available. Matching a Data Source Factory service to a Persistence Unit is discussed in *Database Access* on page 591.

A JPA Provider must track the life cycle of the matching Data Source Factory service; while this service is unavailable the Entity Manager Factory service must also be unavailable. Any active Entity Managers created by the Entity Manager Factory service become invalid to use at that time.

The Entity Manager Factory service must be registered with the same service properties as described for the Entity Manager Factory Builder service, see *Service Registrations* on page 589. It should be registered under the following name:

```
javax.persistence.EntityManagerFactory
```

The `EntityManagerFactory` interface is from the JPA packages and is therefore bound to one of the two supported versions, see *Dependencies* on page 580.

An Entity Manager Factory is bound to a Data Source Factory service because its assigned Persistence Unit was complete. However, a Client Bundle could still provide JDBC configuration properties for the `createEntityManager(Map)` method. This not always possible, see *Rebinding* on page 592.

In the case of an incomplete Persistence Unit no Entity Manager Factory can be initially registered, however once configured using an Entity Manager Factory Builder service the JPA Service must register the created Entity Manager Factory as a service. The registered service must include any supplied configuration properties that match the recommended OSGi service property types as service properties. The `javax.persistence.jdbc.password` property must be omitted from these service properties.

If the Entity Manager Factory Builder service is later used to change the configuration being used by the Entity Manager Factory Service then the registered Entity Manager Factory service must be unregistered and closed. The newly created Entity Manager Factory object must then be registered as a service.

127.4.8 Stopping

If a Persistence Bundle is being stopped, then the JPA Provider must ensure that any resources allocated on behalf of the Persistence Bundle are cleaned up and all open connections are closed. This cleanup must happen synchronously with the `STOPPING` event. Any Exceptions being thrown while cleaning up should be logged but must not stop any further clean up.

If the JPA Provider is being stopped, the JPA Provider must unregister all JPA Services that it registered through the Persistence Bundles and clean up as if those bundles were stopped.

127.4.9 Entity Manager Factory Life Cycle

The Entity Manager Factory object has a `close` method. This method closes the EntityManagerFactory and all associated Entity Manager instances. As an OSGi framework is a multi-tenant environment it should not be possible for one user of an Entity Manager Factory service to break the valid usage of another. Therefore calls to the `close` method of the EntityManagerFactory registered in the service registry *must not* close the Entity Manager Factory.

When an Entity Manager Factory Builder service is used to create an Entity Manager Factory the same rules apply to the resulting Entity Manager Factory service, however the object returned by the Entity Manager Factory Builder behaves differently. This object has a working close method which must unregister the Entity Manager Factory service and close the Entity Manager Factory. This allows callers of the Entity Manager Factory Builder to invalidate the Entity Manager Factories that they create if, for example, a configuration changes, or a Data Source becomes invalid.

127.5 JPA Provider

JPA Providers supply the implementation of the JPA Services and the Persistence Provider service. It is the responsibility of a JPA Provider to store and retrieve the entity classes from a relational database. It is the responsibility of the JPA Provider to register a Persistence Provider and start tracking Persistence Bundles, see *Extending a Persistence Bundle* on page 587.

127.5.1 Managed Model

A JPA Provider that supports running in managed mode should register a specific service for the Java EE Containers: the Persistence Provider service. The interface is the standard JPA `PersistenceProvider` interface. See *Dependencies* on page 580 for the issues around the multiple versions that this specification supports.

The service must be registered with the following service property:

- `javax.persistence.provider` - The JPA Provider implementation class name, a documented name for all JPA Providers.

The Persistence Provider service enables a Java EE Container to find a particular JPA Provider. This service is intended for containers only, not for Client Bundles because there are implicit assumptions in the JPA Providers about the Java EE environment. A Java EE Container must obey the life cycle of the Persistence Provider service. If this service is unregistered then it must close all connections and clean up the corresponding resources.

127.5.2 Database Access

A Persistence Unit is configured to work with a relational database. JPA Providers must communicate with a relational database through a compliant JDBC database driver. The database and driver parameters are specified with properties in the Persistence Unit or the configuration properties when a Entity Manager Factory Builder is used to build an Entity Manager Factory. All JPA Providers, regardless of version, in an OSGi environment must support the following properties for database access:

- `javax.persistence.jdbc.driver` - Fully-qualified name of the driver class.
- `javax.persistence.jdbc.url` - Driver-specific URL to indicate database information
- `javax.persistence.jdbc.user` - User name to use when obtaining connections
- `javax.persistence.jdbc.password` - Password to use when obtaining connections

There are severe limitations in specifying these properties after the Entity Manager Factory is created for the first time, see *Rebinding* on page 592.

127.5.3 Data Source Factory Service Matching

Providers must use the `javax.persistence.jdbc.driver` property, as defined in *JDBC Access in JPA* on page 584, to obtain a Data Source Factory service. The Data Source Factory is specified in *Data Ser-*

vice Specification for JDBC™ Technology on page 551. The javax.persistence.jdbc.driver property must be matched with the value of the Data Source Factory service property named osgi.jdbc.driver.class.

The Data Source Factory service is registered with the osgi.jdbc.driver.class service property that holds the class name of the driver. This property must match the javax.persistence.jdbc.driver service property of the Persistence Unit.

For example, if the Persistence Unit specifies the com.acme.db.Driver database driver in the javax.persistence.jdbc.driver property (or in the Persistence Descriptor property element), then the following filter would select an appropriate Data Source Factory:

```
(&(objectClass=org.osgi.service.jdbc.DataSourceFactory)
   (osgi.jdbc.driver.class=com.acme.db.Driver))
```

Once the Data Source Factory is obtained, the JPA Provider must obtain a DataSource object. This Data Source object must then be used for all relational database access.

In [1] *JPA 1.0* the JPA JDBC properties were not standardized. JPA Providers typically defined a set of JDBC properties, similar to those defined in JPA 2.0, to configure JDBC driver access. JPA 1.0 JPA Providers must look up the Data Source Factory service first using the JPA 2.0 JDBC properties. If these properties are not defined then they should fall back to their proprietary driver properties.

127.5.4 Rebinding

In this specification, the Entity Manager Factory service is only registered when the Persistence Unit is complete and a matching Data Source Factory service is available. However, the API of the Entity Manager Factory Builder allows the creation of an Entity Manager Factory with configuration properties. Those configuration properties could contain the JDBC properties to bind to another Data Source Factory service than it had already selected.

This case must not be supported by a JPA Provider, an Illegal Argument Exception must be thrown. If such a case would be supported then the life cycle of the Entity Manager Factory service would still be bound to the first Data Source Factory. There would be no way for the JPA Provider to signal to the Client Bundle that the returned Entity Manager Factory is no longer valid because the rebound Data Source Factory was unregistered.

Therefore, when an Entity Manager Factory is being created using the Entity Manager Factory Builder, a JPA Provider must verify that the new properties are compatible with the properties of the already created Entity Manager Factory. If no, then an Exception must be thrown. If they are compatible, then an instance of the previous Entity Manager Factory should be returned.

127.5.5 Enhancing Entity Classes

JPA Providers may choose to implement the JPA specifications using various implementation approaches and techniques. This promotes innovation in the area, but also opens the door to limitations and constraints arising due to implementation choices. For example, there are JPA Providers that perform byte code weaving during the entity class loading. Dynamic byte code weaving requires that the entity classes are not loaded until the JPA Provider is first able to intercept the loading of the entity class and be given an opportunity to do its weaving. It also implies that the Persistence Bundle and any other bundles that import packages from that bundle must be refreshed if the JPA Provider needs to be changed.

This is necessary because the JPA Services are registered against the Bundle Contexts of the Persistence Bundles and not the Bundle Context of the JPA Providers. Client Bundles must then unget the service to unbind themselves from the uninstalled JPA Provider. However, since most JPA Providers perform some kind of weaving or class transformation on the entity classes, the Persistence Bundle will likely need to be refreshed. This will cause the Client Bundles to be refreshed also because they depend on the packages of the entity classes.

127.5.6 Class Loading

JPA Providers cannot have package dependencies on entity classes in Persistence Bundles because they cannot know at install time what Persistence Bundles they will be servicing. However, when a JPA Provider is servicing a Persistence Bundle, it must be able to load classes and resources from that Persistence Bundle according to the OSGi bundle rules. To do this class loading it must obtain a class loader that has the same visibility as the Persistence Bundle's bundle class loader. This will also allow it to load and manage metadata for the entity classes and resources for that Persistence Bundle's assigned Persistence Units. These resources and entity classes must reside directly in the Persistence Bundle, they must be accessed using the `getEntry` method. Entity classes and resources must not reside in fragments.

127.5.7 Validation

There is not yet an OSGi service specification defined for validation providers. If validation is required, the validation implementation will need to be included with the JPA Provider bundle.

127.6 Static Access

Non-managed client usage of JPA has traditionally been achieved through the `Persistence` class. Invoking a static method on the `Persistence` class is a dependency on the returned JPA Provider that cannot be managed by the OSGi framework.

However, such an unmanaged dependency is supported in this specification by the Static Persistence bundle. This bundle provides backwards compatibility for programs that use existing JPA access patterns. However, usage of this static model requires that the deployer ensures that the actors needed are in place at the appropriate times by controlling the life cycles of all participating bundles. The normal OSGi safe-guards and dependency handling do not work in the case of static access.

A Static Persistence Bundle must provide static access from the `Persistence` class to the JPA Services.

127.6.1 Access

There are two methods on the `Persistence` class:

- `createEntityManagerFactory(String)`
- `createEntityManagerFactory(String,Map)`

Both methods take the name of a Persistence Unit. The last method also takes a map that contains extra configuration properties. To support the usage of the static methods on the `Persistence` class, the implementation of the `Persistence.createEntityManagerFactory` method family must do a lookup of one of the JPA Services associated with the selected Persistence Unit.

If no configuration properties are specified, the Static Persistence Bundle must look for an Entity Manager Factory service with the `osgi.unit.name` property set to the given name. The default service should be used because no selector for a version is provided. If no such service is available, `null` must be returned. Provisioning of multiple versioned Persistence Units is not supported. Deployers should ensure only a single version of a Persistence Unit with the same name is present in an OSGi framework at any moment in time.

Otherwise, if configuration properties are provided, the Static Access implementation must look for an Entity Manager Factory Builder service with the `osgi.unit.name` property set to the given Persistence Unit name. If no such service exists, `null` must be returned. Otherwise, the default service must be used to create an Entity Manager Factory with the given configuration properties. The result must be returned to the caller.

For service lookups, the Static Persistence Bundle must use its own Bundle Context, it must not attempt to use the Bundle Context of the caller. All exceptions should be passed to the caller.

The class space of the Entity Manager Factory and the class space of the client cannot be enforced to be consistent by the framework because it is the Persistence class that is doing the lookup of the service, and not the actual calling Client Bundle that will be using the Entity Manager Factory. The framework cannot make the connection and therefore cannot enforce that the class spaces correspond. Deployers should therefore ensure that the involved class spaces are correctly wired.

127.7 Capabilities

The JPA Service Implementation must supply a number of capabilities for use by client bundles and Deployers.

127.7.1 The Extender Capability

A JPA Service implementation must provide an extender which finds and extends persistence bundles. The bundle providing this extender must provide a capability in the osgi.extender namespace declaring an extender with the name JPA_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.jpa and javax.persistence packages. For example:

```
Provide-Capability: osgi.extender;
    osgi.extender="osgi.jpa";
    version:Version="1.1";
    uses:="org.osgi.service.jpa,javax.persistence"
```

This capability must follow the rules defined for the *osgi.extender Namespace* on page 707.

All persistence bundles should require the osgi.extender capability from the JPA Service. This requirement will wire the persistence bundle to the JPA Service implementation and ensure that the JPA service is using the same API packages as the persistence bundle.

```
Require-Capability: osgi.extender;
  filter:="(&(osgi.extender=osgi.jpa)(version>=1.1)(!(version>=2.0)))"
```

This requirement can be easily generated using the RequireJPAExtender annotation.

The JPA extender must only process a persistence bundle's persistence units if the following is true:

- The bundle's wiring has a required wire for at least one osgi.extender capability with the name osgi.jpa and the first of these required wires is wired to the JPA extender.
- The bundle's wiring has no required wire for an osgi.extender capability with the name osgi.jpa.

Otherwise, the JPA Service extender must not process the persistence bundle

127.7.2 The JPA Contract Capability

Previous versions of this specification recommended that the JPA API packages were versioned using the OSGi recommended semantic versioning policy. Whilst this would have been an excellent way to ensure compatibility between JPA persistence bundles, client bundles, and JPA providers, in practice few bundles followed this versioning policy. As a result the various actors in the JPA service can easily be created with have clashing version ranges.

This problem is not isolated to JPA, and so a general solution was created called [5] *Portable Java Contract Definitions*. These define a capability namespace called osgi.contract

In order to permit JPA clients to reliably work when paired with newer versions of JPA there needs to be a defined contract upon which the clients and persistence units can rely, otherwise a JPA 1.0 compatible client cannot declare a dependency which also accepts the backward compatible JPA 2.0 API. For JPA the following three contracts exist:

```
osgi.contract;osgi.contract=JavaJPA;version:Version=1;
    uses:="javax.persistence,javax.persistence.spi"
```

```
osgi.contract;osgi.contract=JavaJPA;version:Version=2;
    uses:="javax.persistence,javax.persistence.criteria,
    javax.persistence.metamodel,javax.persistence.spi"
```

```
osgi.contract;osgi.contract=JavaJPA;version:Version=2.1;
          uses:="javax.persistence,javax.persistence.criteria,
          javax.persistence.metamodel,javax.persistence.spi"
```

JPA API providers must declare the full set of API contract versions with which they are compatible. As JPA API versions are backward compatible this will typically result in the provider exposing all versions of a contract. Note that when a provider offers multiple versions of a contract then all of the contract versions must be offered by a single capability. For example:

```
Export-Package: javax.persistence,javax.persistence.criteria,
 javax.persistence.metamodel,javax.persistence.spi
Provide-Capability: osgi.contract;osgi.contract=JavaJPA;
 version:List>Version<="2.1,2,1"; uses:="javax.persistence,
 javax.persistence.criteria,javax.persistence.metamodel,javax.persistence.spi"
```

The contract capability means that clients can safely import the API using the contract and no import version. For example:

```
Import-Package: javax.persistence,javax.persistence.criteria
Require-Capability: osgi.contract;
 filter:="(&(osgi.contract=JavaJPA)(version=2.1))"
```

127.7.3 Service capabilities

The JPA Service implementation is responsible for registering both an EntityManagerFactoryBuilder service and a EntityManagerFactory service on behalf of the persistence bundle. The persistence bundle should therefore provide two capabilities in the osgi.service namespace, one representing the EntityManagerFactoryBuilder service, and another representing the javax.persistence.EntityManagerFactory service. These capabilities must also declare uses constraints for the packages that they expose. For example:

```
Provide-Capability: osgi.service;
    objectClass:List<String>=
      "org.osgi.service.jpa.EntityManagerFactoryBuilder";
    uses:="org.osgi.service.jpa",
    osgi.service;objectClass:List<String>=
      "javax.persistence.EntityManagerFactory";
    uses:="javax.persistence"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

127.8 Security

When Java permissions are enabled, the JPA service must perform the following security procedures.

127.8.1 Service Permissions

The JPA service is built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish services. A persistence bundle therefore must have ServicePermission[<interface>, REGISTER] for both the EntityManagerFactory and EntityManagerFactoryBuilder services.

If a persistence bundle specifies a complete persistence unit then the persistence bundle must either have ServicePermission[<org.osgi.service.jdbc.DataSourceFactory>, GET], or be able to directly load the configured database driver.

Client bundles that wish to configure a persistence unit using the EntityManagerFactoryBuilder service must have ServicePermission[<org.osgi.service.jpa.EntityManagerFactoryBuilder>, GET]. Furthermore, if this service is used to configure an incomplete persistence unit with a database driver name then it is the getter of the EntityManagerFactoryBuilder service whose permissions must be checked when obtaining the DataSourceFactory service. If the caller of the EntityManagerFactory Builder passes a ready constructed database Driver or DataSource then no permission check is required.

127.8.2 Required Admin Permission

The JPA service implementation requires AdminPermission[*,CONTEXT] because it needs access to the bundle's Bundle Context object with the Bundle.getBundleContext() method.

127.9 org.osgi.service.jpa

JPA Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jpa; version="[1.1,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jpa; version="[1.1,1.2)"
```

127.9.1 Summary

- EntityManagerFactoryBuilder - This service interface offers JPA clients the ability to create instances of EntityManagerFactory for a given named persistence unit.

127.9.2 public interface EntityManagerFactoryBuilder

This service interface offers JPA clients the ability to create instances of EntityManagerFactory for a given named persistence unit. A service instance will be created for each named persistence unit and can be filtered by comparing the value of the osgi.unit.name property containing the persistence unit name. This service is used specifically when the caller wants to pass in factory-scoped properties as arguments. If no properties are being used in the creation of the EntityManagerFactory then the basic EntityManagerFactory service should be used.

Provider Type Consumers of this API must not implement this type

127.9.2.1 public static final String JPA_CAPABILITY_NAME = "osgi.jpa"

The name of the JPA extender capability.

Since 1.1

127.9.2.2 public static final String JPA_SPECIFICATION_VERSION = "1.1"

The version of the extender capability for the JPA Service specification

Since 1.1

127.9.2.3 public static final String JPA_UNIT_NAME = "osgi.unit.name"

The name of the persistence unit.

127.9.2.4 public static final String JPA_UNIT_PROVIDER = "osgi.unit.provider"

The class name of the provider that registered the service and implements the JPA javax.persistence.PersistenceProvider interface.

127.9.2.5 public static final String JPA_UNIT_VERSION = "osgi.unit.version"

The version of the persistence unit bundle.

127.9.2.6 public EntityManagerFactory createEntityManagerFactory(Map<String, Object> props)

props Properties to be used, in addition to those in the persistence descriptor, for configuring the EntityManagerFactory for the persistence unit.

□ Return an EntityManagerFactory instance configured according to the properties defined in the corresponding persistence descriptor, as well as the properties passed into the method.

Returns An EntityManagerFactory for the persistence unit associated with this service. Must not be null.

127.9.2.7 public Bundle getPersistenceProviderBundle()

□ This method returns the Bundle which provides the PersistenceProvider implementation that is used by this EntityManagerFactoryBuilder.

If the PersistenceProvider is provided as an OSGi service then this method must return the bundle which registered the service. Otherwise this method must return the bundle which loaded the PersistenceProvider implementation class.

Returns The Bundle which provides the PersistenceProvider implementation used by this EntityManagerFactoryBuilder.

Since 1.1

127.9.2.8 public String getPersistenceProviderName()

□ This method returns the name of the PersistenceProvider implementation that is used by this EntityManagerFactoryBuilder. The returned value will be the same as the value of the JPA_UNIT_PROVIDER service property.

Returns the name of the PersistenceProvider implementation

Since 1.1

127.10 org.osgi.service.jpa.annotations

JPA Service Annotations Package Version 1.1.

This package contains annotations that can be used to require the JPA Service implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

127.10.1 Summary

- `RequireJPAExtender` - This annotation can be used to require the JPA extender.

127.10.2 @RequireJPAExtender

This annotation can be used to require the JPA extender. It can be used directly, or as a meta-annotation.

Retention `CLASS`

Target `TYPE`, `PACKAGE`

127.11 References

[1] *JPA 1.0*
https://jcp.org/en/jsr/detail?id=220

[2] *JPA 2.0*
https://jcp.org/en/jsr/summary?id=317

[3] *JPA 2.1*
https://jcp.org/en/jsr/summary?id=317

[4] *Java EE 5*
https://www.oracle.com/java/technologies/javaee/javaeetechnologies.html#javaee5

[5] *Portable Java Contract Definitions*
https://docs.osgi.org/reference/portable-java-contracts.html

128 Web Applications Specification

Version 1.0

128.1 Introduction

The Java EE Servlet model has provided the backbone of web based applications written in Java. Given the popularity of the Servlet model, it is desirable to provide a seamless experience for deploying existing and new web applications to Servlet containers operating on the OSGi framework. Previously, the Http Service in the catalog of OSGi compendium services was the only model specified in OSGi to support the Servlet programming model. However, the Http Service, as defined in that specification, is focused on the run time, as well as manual construction of the servlet context, and thus does not actually support the standard Servlet packaging and deployment model based on the Web Application Archive, or WAR format.

This specification defines the Web Application Bundle, which is a bundle that performs the same role as the WAR in Java EE. A WAB uses the OSGi life cycle and class/resource loading rules instead of the standard Java EE environment. WABs are normal bundles and can leverage the full set of features of the OSGi framework.

Web applications can also be installed as traditional WARs through a manifest rewriting process. During the install, a WAR is transformed into a WAB. This specification was based on ideas developed in [5] *PAX Web Extender*.

This Web Application Specification provides support for web applications written to the Servlet 2.5 specification, or later. Given that Java Server Pages, or JSPs, are an integral part of the Java EE web application framework, this specification also supports the JSP 2.1 specification or greater if present. This specification details how a web application packaged as a WAR may be installed into an OSGi framework, as well as how this application may interact with, and obtain, OSGi services.

128.1.1 Essentials

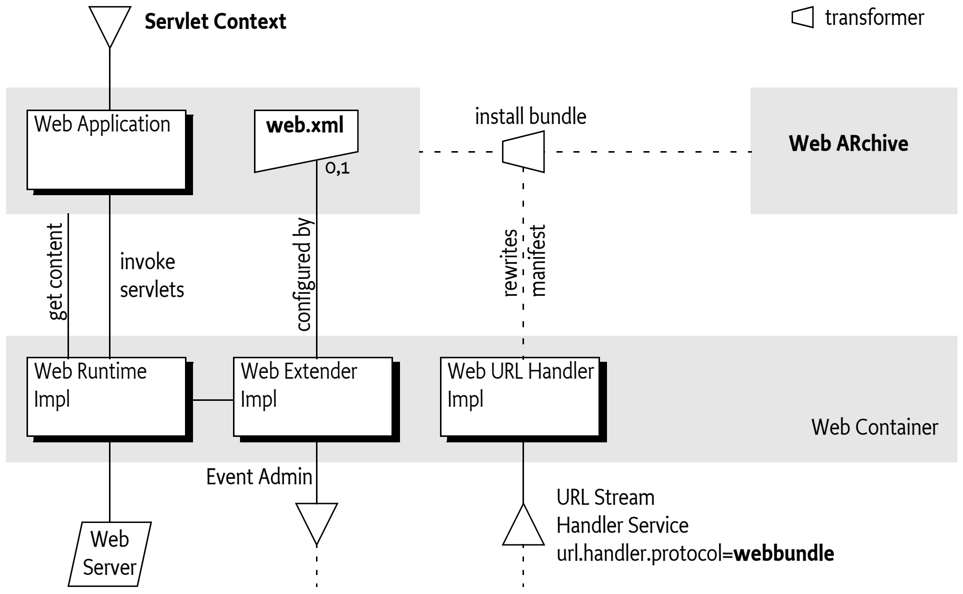
- *Extender* - Enable the configuration of components inside a bundle based on configuration data provided by the bundle developer.
- *Services* - Enable the use of OSGi services within a Web Application.
- *Deployment* - Define a mechanism to deploy Web Applications, both OSGi aware and non OSGi aware, in the OSGi environment.
- *WAR File Support* - Transparently enhance the contents of a WAR's manifest during installation to add any headers necessary to deploy a WAR as an OSGi bundle.

128.1.2 Entities

- *Web Container* - The implementation of this specification. Consists of a Web Extender, a Web URL Handler and a Servlet and Java Server Pages Web Runtime environment.
- *Web Application* - A program that has web accessible content. A Web Application is defined by [2] *Java EE Web Applications*.
- *Web Application Archive (WAR)* - The Java EE standard resource format layout of a JAR file that contains a deployable Web Application.
- *Web Application Bundle* - A Web Application deployed as an OSGi bundle, also called a WAB.
- *WAB* - The acronym for a Web Application Bundle.

- *Web Extender* - An extender bundle that deploys the Web Application Bundle to the Web Runtime based on the Web Application Bundle's state.
- *Web URL Handler* - A URL handler which transforms a Web Application Archive (WAR) to conform to the OSGi specifications during installation by installing the WAR through a special URL so that it becomes a Web Application Bundle.
- *Web Runtime* - A Java Server Pages and Servlet environment, receiving the web requests and translating them to servlet calls, either from Web Application servlets or other classes.
- *Web Component* - A Servlet or Java Server Page (JSP).
- *Servlet* - An object implementing the Servlet interface; this is for the request handler model in the Servlet Specification.
- *Servlet Context* - The model representing the Web Application in the Servlet Specification.
- *Java Server Page (JSP)* - A declarative, template based model for generating content through Servlets that is optionally supported by the Web Runtime.
- *Context Path* - The URI path prefix of any content accessible in a Web Application.

Figure 128.1 *Web Container Entities*



128.1.3 Dependencies

The package dependencies for the clients of this specification are listed in the following table.

Table 128.1 *Dependency versions*

Packages	Export Version	Client Import Range
javax.servlet	2.5	[2.5,3.0)
javax.servlet.http	2.5	[2.5,3.0)
javax.servlet.jsp.el	2.1	[2.1,3.0)
javax.servlet.jsp.jstl.core	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.fmt	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.sql	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.tlv	1.2	[1.2,2.0)
javax.servlet.jsp.resources	2.1	[2.1,3.0)

Packages	Export Version	Client Import Range
javax.servlet.jsp.tagext	2.1	[2.1,3.0)
javax.servlet.jsp	2.1	[2.1,3.0)

JSP is optional for the Web Runtime.

128.1.4 Synopsis

The Web Application Specification is composed of a number of cooperating parts, which are implemented by a *Web Container*. A Web Container consists of:

- *Web Extender* - Responsible for deploying Web Application Bundles (WAB) to a Web Runtime,
- *Web Runtime* - Provides support for Servlet and optionally for JSPs, and
- *Web URL Handler* - Provides on-the-fly enhancements of non-OSGi aware Web ARchives (WAR) so that they can be installed as a WAB.

WABs are standard OSGi bundles with additional headers in the manifest that serve as deployment instructions to the Web Extender. WABs can also contain the Java EE defined web.xml descriptor in the WEB-INF/ directory. When the Web Extender detects that a WAB is ready the Web Extender deploys the WAB to the Web Runtime using information contained in the web.xml descriptor and the appropriate manifest headers. The Bundle Context of the WAB is made available as a Servlet Context attribute. From that point, the Web Runtime will use the information in the WAB to serve content to any requests. Both dynamic as well as static content can be provided.

The Web URL Handler allows the deployment of an unmodified WAR as a WAB into the OSGi framework. This Web URL Handler provides a URL stream handler with the webbundle: scheme. Installing a WAR with this scheme allows the Web URL Handler to interpose itself as a filter on the input stream of the contents of the WAR, transforming the contents of the WAR into a WAB. The Web URL Handler rewrites the manifest by adding necessary headers to turn the WAR into a valid WAB. Additional headers can be added to the manifest that serve as instructions to the Web Extender.

After a WAB has been deployed to the Web Runtime, the Web Application can interact with the OSGi framework via the provided Bundle Context. The Servlet Context associated with this WAB follows the same life cycle as the WAB. That is, when the underlying Web Application Bundle is started, the Web Application is deployed to the Web Runtime. When the underlying Web Application Bundle is stopped because of a failure or other reason, the Web Application is undeployed from the Web Run-time.

128.2 Web Container

A Web Container is the implementation of this specification. It consists of the following parts:

- *Web Extender* - Detects Web Application Bundles (WAB) and tracks their life cycle. Ready WABs are deployed to the Web Runtime.
- *Web Runtime* - A runtime environment for a Web Application that supports the [3] *Servlet 2.5 specification* and [4] *JSP 2.1 specification* or later. The Web Runtime receives web requests and calls the appropriate methods on servlets. Servlets can be implemented by classes or Java Server Pages.
- *Web URL Handler* - A URL stream handler providing the webbundle: scheme. This scheme can be used to install WARs in an OSGi framework. The Web URL Handler will then automatically add the required OSGi manifest headers.

The extender, runtime, and handler can all be implemented in the same or different bundles and use unspecified mechanisms to communicate. This specification uses the defined names of the subparts as the actor; the term Web Container is the general name for this collection of actors.

128.3 Web Application Bundle

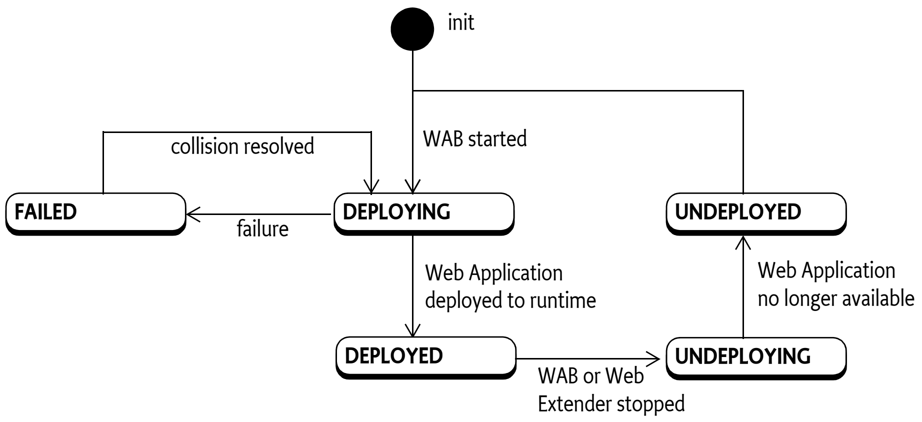
Bundles are the deployment and management entities under OSGi. A *Web Application Bundle* (WAB) is deployed as an OSGi bundle in an OSGi framework, where each WAB provides a single *Web Application.* A Web Application can make use of the [3] *Servlet 2.5 specification* and [4] *JSP 2.1 specification* programming models, or later, to provide content for the web.

A WAB is defined as a normal OSGi bundle that contains web accessible content, both static and dynamic. There are no restrictions on bundles. A Web Application can be packaged as a WAB during application development, or it can be transparently created at bundle install time from a standard Web Application aRchive (WAR) via transformation by the Web URL Handler, see *Web URL Handler* on page 606.

A WAB is a valid OSGi bundle and as such must fully describe its dependencies and exports (if any). As Web Applications are modularized further into multiple bundles (and not deployed as WAR files only) it is possible that a WAB can have dependencies on other bundles.

A WAB may be installed into the framework using the `BundleContext.installBundle` methods. Once installed, a WAB's life cycle is managed just like any other bundle in the framework. This life cycle is tracked by the Web Extender who will then deploy the Web Application to the Web Runtime when the WAB is ready and will undeploy it when the WAB is no longer ready. This state is depicted in Figure 128.2.

Figure 128.2 *State diagram Web Application*


init
WAB started
collision resolved
FAILED
failure
DEPLOYING
UNDEPLOYED
Web Application deployed to runtime
Web Application no longer available
DEPLOYED
UNDEPLOYING
WAB or Web Extender stopped

128.3.1 WAB Definition

A WAB is differentiated from non Web Application bundles through the specification of the additional manifest header:

```
Web-ContextPath ::= path
```

The Web-ContextPath header specifies the value of the *Context Path* of the Web Application. All web accessible content of the Web Application is available on the web server relative to this Context Path. For example, if the context path is `/sales`, then the URL would be something like: `http://www.acme.com/sales`. The Context Path must always begin with a solidus ('/' `\u002F`).

The Web Extender must not recognize a bundle as a Web Application unless the Web-ContextPath header is present in its manifest and the header value is a valid path for the bundle.

A WAB can optionally contain a `web.xml` resource to specify additional configuration. This `web.xml` must be found with the Bundle `findEntries` method at the path:

```
WEB-INF/web.xml
```

The findEntries method includes fragments, allowing the web.xml to be provided by a fragment. The Web Extender must fully support a web.xml descriptor that specifies Servlets, Filters, or Listeners whose classes are required by the WAB.

128.3.2 Starting the Web Application Bundle

A WAB's Web Application must be *deployed* while the WAB is *ready*. Deployed means that the Web Application is available for web requests. Once deployed, a WAB can serve its web content on the given Context Path. Ready is when the WAB:

- Is in the ACTIVE state, or
- Has a lazy activation policy and is in the STARTING state.

The Web Extender should ensure that serving static content from the WAB does not activate the WAB when it has a lazy activation policy.

To deploy the WAB, the Web Extender must initiate the deploying of the Web Application into a Web Runtime. This is outlined in the following steps:

1. Wait for the WAB to become ready. The following steps can take place asynchronously with the starting of the WAB.
2. Post an org/osgi/service/web/DEPLOYING event. See *Events* on page 609.
3. Validate that the Web-ContextPath manifest header does not match the Context Path of any other currently deployed web application. If the Context Path value is already in use by another Web Application, then the Web Application must not be deployed, and the deployment fails, see *Failure* on page 604. The Web Extender should log the collision. If the prior Web Application with the same Context Path is undeployed later, this Web Application should be considered as a candidate, see *Stopping the Web Application Bundle* on page 605.
4. The Web Runtime processes deployment information by processing the web.xml descriptor, if present. The Web Container must perform the necessary initialization of Web Components in the WAB as described in the [3] *Servlet 2.5 specification.* This involves the following sub-steps in the given order:
   - Create a Servlet Context for the Web Application.
   - Instantiate configured Servlet event listeners.
   - Instantiate configured application filter instances etc.

   The Web Runtime is required to complete instantiation of listeners prior to the start of execution of the first request into the Web Application by the Web Runtime. Attribute changes to the Servlet Context and Http Session objects can occur concurrently. The Servlet Container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.

   If event listeners or filters are used in the web.xml, then the Web Runtime will load the corresponding classes from the bundle activating the bundle if it was lazily started. Such a configuration will therefore not act lazily.
5. Publish the Servlet Context as a service with identifying service properties, see *Publishing the Servlet Context* on page 604.
6. Post an org/osgi/service/web/DEPLOYED event to indicate that the web application is now available. See *Events* on page 609.

If at any moment before the org/osgi/service/web/DEPLOYED event is published the deployment of the WAB fails, then the WAB deployment fails, see *Failure* on page 604.

128.3.3 Failure

Any validation failures must prevent the Web Application from being accessible via HTTP, and must result in a `org/osgi/service/web/FAILED` event being posted. See *Events* on page 609. The situation after the failure must be as if the WAB was never deployed.

128.3.4 Publishing the Servlet Context

To help management agents with tracking the state of Web Applications, the Web Extender must register the Servlet Context of the WAB as a service, using the Bundle Context of the WAB. The Servlet Context service must be registered with the service properties listed in the following table.

Table 128.2 Servlet Context Service Properties

Property Name	Type	Description
`osgi.web.symbolicname`	`String`	The symbolic name for the Web Application Bundle
`osgi.web.version`	`String`	The version of the Web Application Bundle. If no Bundle-Version is specified in the manifest then this property must not be set.
`osgi.web.contextpath`	`String`	The Context Path from which the WAB's content will be served.

128.3.5 Static Content

A deployed WAB provides content on requests from the web. For certain access paths, this can serve content from the resources of the web application: this is called *static content.* A Web Runtime must use the Servlet Context resource access methods to service static content, the resource loading strategy for these methods is based on the `findEntries` method, see *Resource Lookup* on page 610. For confidentiality reasons, a Web Runtime must not return any static content for paths that start with one of the following prefixes:

```
WEB-INF/
OSGI-INF/
META-INF/
OSGI-OPT/
```

These *protected directories* are intended to shield code content used for dynamic content generation from accidentally being served over the web, which is a potential attack route. In the servlet specification, the `WEB-INF/` directory in the WAR is protected in such a way. However, this protection is not complete. A dependent JAR can actually be placed outside the WEB-INF directory that can then be served as static content. The same is true for a WAB. Though the protected directories must never be served over the web, there are no other checks required to verify that no content can be served that is also available from the Bundle class path.

It is the responsibility of the author of the WAB to ensure that confidential information remains confidential by placing it in one of the protected directories. WAB bundles should be constructed in such a way that they do not accidentally expose code or confidential information. The simplest way to achieve this is to follow the WAR model where code is placed in the `WEB-INF/classes` directory and this directory is placed on the Bundle's class path as the first entry. For example:

```
Bundle-ClassPath: WEB-INF/classes, WEB-INF/lib/acme.jar
```

128.3.6 Dynamic Content

Dynamic content is content that uses code to generate the content, for example a servlet. This code must be loaded from the bundle with the Bundle `loadClass` method, following all the Bundle class path rules.

Unlike a WAR, a WAB is not constrained to package classes and code resources in the WEB-INF/classes directory or dependent JARs in WEB-INF/lib/ only. These entries can be packaged in any way that's valid for an OSGi bundle as long as such directories and JARs are part of bundle class path as set with the Bundle-ClassPath header and any attached fragments. JARs that are specified in the Bundle-ClassPath header are treated like JARs in the WEB-INF/lib/ directory of the Servlet specification. Similarly, any directory that is part of the Bundle-ClassPath header is treated like WEB-INF/classes directory of the Servlet specification.

Like WARs, code content that is placed outside the protected directories can be served up to clients as static content.

128.3.7 Content Serving Example

This example consists of a WAB with the following contents:

```
acme.jar:
    Bundle-ClassPath: WEB-INF/classes, LIB/bar.jar
    Web-ContextPath: /acme

    WEB-INF/lib/foo.jar
    LIB/bar.jar
    index.html
    favicon.ico
```

The content of the embedded JARs foo.jar and bar.jar is:

```
foo.jar:                bar.jar:
    META-INF/foo.tld        META-INF/bar.tld
    foo/FooTag.class        bar/BarTag.class
```

Assuming there are no special rules in place then the following lists specifies the result of a number of web requests for static content:

```
/acme/index.html            acme.wab:index.html
/acme/favicon.ico           acme.wab:favicon.ico
/acme/WEB-INF/lib/foo.jar   not found because protecteddirectory
/acme/LIB/bar.jar           acme.wab:LIB/bar.jar  (code, but not protected)
```

In this example, the tag classes in bar.jar must be found (if JSP is supported) but the tag classes in foo.jar must not because foo.jar is not part of the bundle class path.

128.3.8 Stopping the Web Application Bundle

A web application is stopped by stopping the corresponding WAB. In response to a WAB STOPPING event, the Web Extender must *undeploy* the corresponding Web Application from the Servlet Container and clean up any resources. This undeploying must occur synchronously with the WAB's stopping event. This will involve the following steps:

1. An org/osgi/service/web/UNDEPLOYING event is posted to signal that a Web Application will be removed. See *Events* on page 609.
2. Unregister the corresponding Servlet Context service
3. The Web Runtime must stop serving content from the Web Application.
4. The Web Runtime must clean up any Web Application specific resources as per servlet 2.5 specification.
5. Emit an org/osgi/service/web/UNDEPLOYED event. See *Events* on page 609.
6. It is possible that there are one or more *colliding* WABs because they had the same Context Path as this stopped WAB. If such colliding WABs exists then the Web Extender must attempt to deploy the colliding WAB with the lowest bundle id.

Any failure during undeploying should be logged but must not stop the cleaning up of resources and notification of (other) listeners as well as handling any collisions.

128.3.9 Uninstalling the Web Application Bundle

A web application can be uninstalled by uninstalling the corresponding WAB. The WAB will be uninstalled from the OSGi framework.

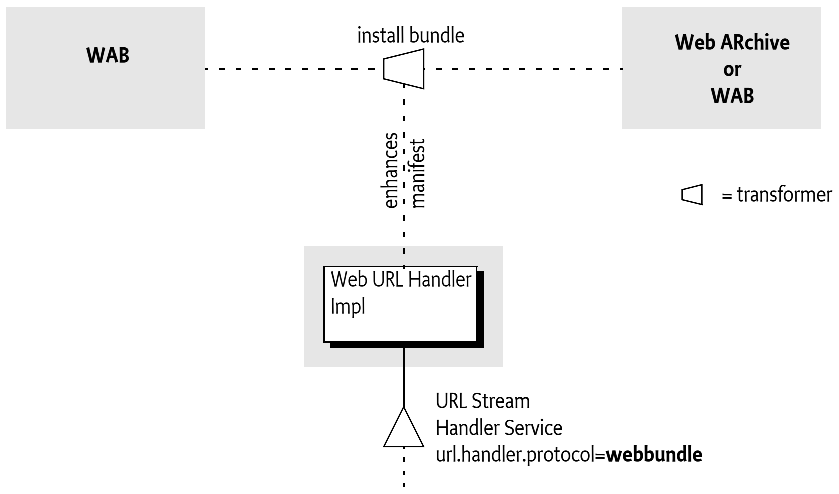
128.3.10 Stopping of the Web Extender

When the Web Extender is stopped all deployed WABs are undeployed as described in *Stopping the Web Application Bundle* on page 605. Although the WAB is undeployed it remains in the ACTIVE state. When the Web Extender leaves the STOPPING state all WABs will have been undeployed.

128.4 Web URL Handler

The Web URL Handler acts as a filter on the Input Stream of an install operation. It receives the WAB or WAR and it then generates a JAR that conforms to the WAB specification by rewriting the manifest resource. This process is depicted in Figure 128.3.

Figure 128.3 Web URL Handler



When the Web Container bundle is installed it must provide the webbundle: scheme to the URL class. The Web URL Handler has two primary responsibilities:

- *WAB* - If the source is already a bundle then only the Web-ContextPath can be set or overwritten.
- *WAR* - If the source is a WAR (that is, it must not contain any OSGi defined headers) then convert the WAR into a WAB.

The Web URL Handler can take parameters from the query arguments of the install URL, see *URL Parameters* on page 607.

The URL handler must validate query parameters, and ensure that the manifest rewriting results in valid OSGi headers. Any validation failures must result in Bundle Exception being thrown and the bundle install must fail.

Once a WAB is generated and installed, its life cycle is managed just like any other bundle in the framework.

128.4.1 URL Scheme

The Web URL Handler's scheme is defined to be:

```
scheme     ::= 'webbundle:' embedded '?' web-params
embedded   ::= <embedded URL according to RFC 1738>
web-params ::= ( web-param ( '&' web-param )* )?
web-param  ::= <key> '=' <value>
```

The web-param <key> and <value> as well as the <embedded url> must follow [6] *Uniform Resource Locators, RFC 1738* for their escaping and character set rules.A Web URL must further follow all the rules of a URL. Whitespaces are not allowed between terms.

An example for a webbundle: URL:

```
webbundle:http://www.acme.com:8021/sales.war?Web-ContextPath=/sales
```

Any URL scheme understood by the framework can be embedded, such as an http:, or file: URL. Some forms of embedded URL also contain URL query parameters and this must be supported. The embedded URL most be encoded as a standard URL. That is, the control characters like colon (':' \u003A), solidus ('/' \u002F), percent ('%' \u0025), and ampersand ('&' \u0026) must not be encoded. Thus the value returned from the getPath method may contain a query part. Any implementation must take care to preserve both the query parameters for the embedded URL, and for the complete webbundle: URL. A question mark must always follow the embedded URL to simplify this processing. The following example shows an HTTP URL with some query parameters:

```
webbundle:http://www.acme.com/sales?id=123?Bundle-SymbolicName=com.example&
                                             Web-ContextPath=/
```

128.4.2 URL Parsing

The URL object for a webbundle: URL must return the following values for the given methods:

- getProtocol - webbundle
- getPath - The complete embedded URL
- getQuery - The parameters for processing of the manifest.

For the following example:

```
webbundle:http://acme.com/repo?war=example.war?Web-ContextPath=/sales
```

The aforementioned methods must return:

- getProtocol - webbundle
- getPath - http://acme.com/repo?war=example.war
- getQuery - Web-ContextPath=/sales

128.4.3 URL Parameters

All the parameters in the webbundle: URL are optional except for the Web-ContextPath parameter. The parameter names are case insensitive, but their values must be treated as case sensitive. Table 128.3 describes the parameters that must be supported by any webbundle: URL Stream handler. A Web URL Handler is allowed to support additional parameters.

Table 128.3 Web bundle URL Parameters

Parameter Name	Description
Bundle-SymbolicName	The desired symbolic name for the resulting WAB.

Parameter Name	Description
Bundle-Version	The version of the resulting WAB. The value of this parameter must follow the OSGi versioning syntax.
Bundle-ManifestVersion	The desired bundle manifest version. Currently, the only valid value for this parameter is 2.
Import-Package	A list of packages that the war file depends on.
Web-ContextPath	The Context Path from which the Servlet Container should serve content from the resulting WAB. This is the only valid parameter when the input JAR is already a bundle. This parameter must be specified.

128.4.4 WAB Modification

The Web URL Handler can set or modify the Web-ContextPath of a WAB if the input source is already a bundle. It must be considered as a bundle when any of the OSGi defined headers listed in Table 128.3 is present in the bundle.

For WAB Modification, the Web URL Handler must only support the Web-ContextPath parameter and it must not modify any existing headers other than the Web-ContextPath. Any other parameter given must result in a Bundle Exception.

128.4.5 WAR Manifest Processing

The Web URL Handler is designed to support the transparent deployment of Java EE Web ARchives (WAR). Such WARs are ignorant of the requirements of the underlying OSGi framework that hosts the Web Runtime. These WARs are not proper OSGi bundles because they do not contain the necessary metadata in the manifest. For example, a WAR without a Bundle-ManifestVersion, Import-Package, and other headers cannot operate in an OSGi framework.

The Web URL Handler implementation copies the contents of the embedded URL to the output and rewrites the manifest headers based on the given parameters. The result must be a WAB.

Any parameters specified must be treated as manifest headers for the web. The following manifest headers must be set to the following values if not specified:

- Bundle-ManifestVersion - Must be set to 2.
- Bundle-SymbolicName - Generated in an implementation specific way.
- Bundle-ClassPath - Must consist of:
  - WEB-INF/classes
  - All JARs from the WEB-INF/lib directory in the WAR. The order of these embedded JARs is unspecified.
  - If these JARs declare dependencies in their manifest on other JARs in the bundle, then these jars must also be appended to the Bundle-ClassPath header. The process of detecting JAR dependencies must be performed recursively as indicated in the Servlet Specification.
- Web-ContextPath - The Web-ContextPath must be specified as a parameter. This Context Path should start with a leading solidus ('/' \u002F). The Web URL handler must add the preceding solidus it if it is not present.

The Web URL Handler is responsible for managing the import dependencies of the WAR. Implementations are free to handle the import dependencies in an implementation defined way. They can augment the Import-Package header with byte-code analysis information, add a fixed set of clauses, and/or use the DynamicImport-Package header as last resort.

Any other manifest headers defined as a parameter or WAR manifest header not described in this section must be copied to the WAB manifest by the Web URL Handler. Such an header must not be modified.

128.4.6 Signed WAR files

When a signed WAR file is installed using the Web URL Handler, then the manifest rewriting process invalidates the signatures in the bundle. The OSGi specification requires fully signed bundles for security reasons, security resources in partially signed bundles are ignored.

If the use of the signing metadata is required, the WAR must be converted to a WAB during development and then signed. In this case, the Web URL Handler cannot be used. If the Web URL Handler is presented with a signed WAR, the manifest name sections that contain the resource's check sums must be stripped out by the URL stream handler. Any signer files (*.SF and their corresponding DSA/RSA signature files) must also be removed.

128.5 Events

The Web Extender must track all WABs in the OSGi framework in which the Web Extender is installed. The Web Extender must post Event Admin events, which is asynchronous, at crucial points in its processing. The topic of the event must be one of the following values:

- org/osgi/service/web/DEPLOYING - The Web Extender has accepted a WAB and started the process of deploying a Web Application.
- org/osgi/service/web/DEPLOYED - The Web Extender has finished deploying a Web Application, and the Web Application is now available for web requests on its Context Path.
- org/osgi/service/web/UNDEPLOYING - The web extender started undeploying the Web Application in response to its corresponding WAB being stopped or the Web Extender is stopped.
- org/osgi/service/web/UNDEPLOYED - The Web Extender has undeployed the Web Application. The application is no longer available for web requests.
- org/osgi/service/web/FAILED - The Web Extender has failed to deploy the Web Application, this event can be fired after the DEPLOYING event has fired and indicates that no DEPLOYED event will be fired.

For each event topic above, the following properties must be published:

- bundle.symbolicName - (String) The bundle symbolic name of the WAB.
- bundle.id - (Long) The bundle id of the WAB.
- bundle - (Bundle) The Bundle object of the WAB.
- bundle.version - (Version) The version of the WAB.
- context.path - (String) The Context Path of the Web Application.
- timestamp – (Long) The time when the event occurred
- extender.bundle - (Bundle) The Bundle object of the Web Extender Bundle
- extender.bundle.id - (Long) The id of the Web Extender Bundle.
- extender.bundle.symbolicName - (String) The symbolic name of the Web Extender Bundle.
- extender.bundle.version - (Version) The version of the Web Extender Bundle.

In addition, the org/osgi/service/web/FAILED event must also have the following property:

- exception - (Throwable) If an exception caused the failure, an exception detailing the error that occurred during the deployment of the WAB.
- collision - (String) If a name collision occurred, the Web-ContextPath that had a collision
- collision.bundles - (Collection<Long>) If a name collision occurred, a collection of bundle ids that all have the same value for the Web-ContextPath manifest header.

128.6 Interacting with the OSGi Environment

128.6.1 Bundle Context Access

In order to properly integrate in an OSGi environment, a Web Application can access the OSGi service registry for publishing its services, accessing services provided by other bundles, and listening to bundle and service events to track the life cycle of these artifacts. This requires access to the Bundle Context of the WAB.

The Web Extender must make the Bundle Context of the corresponding WAB available to the Web Application via the Servlet Context osgi-bundlecontext attribute. A Servlet can obtain a Bundle Context as follows:

```
BundleContext ctxt = (BundleContext)
    servletContext.getAttribute("osgi-bundlecontext");
```

128.6.2 Other Component Models

Web Applications sometimes need to inter-operate with services provided by other component models, such as a Declarative Services or Blueprint. Per the Servlet specification, the Servlet Container owns the life cycle of a Servlet; the life cycle of the Servlet must be subordinate to the life cycle of the Servlet Context, which is only dependent on the life cycle of the WAB. Interactions between different bundles are facilitated by the OSGi service registry. This interaction can be managed in several ways:

- A Servlet can obtain a Bundle Context from the Servlet Context for performing service registry operations.
- Via the JNDI Specification and the osgi:service JNDI namespace. The OSGi JNDI specification describes how OSGi services can be made available via the JNDI URL Context. It defines an osgi:service namespace and leverages URL Context factories to facilitate JNDI integration with the OSGi service registry.

Per this specification, it is not possible to make the Servlet life cycle dependent on the availability of specific services. Any synchronization and service dependency management must therefore be done by the Web Application itself.

128.6.3 Resource Lookup

The getResource and getResourceAsStream methods of the ServletContext interface are used to access resources in the web application. For a WAB, these resources must be found according to the findEntries method, this method includes fragments. For the getResource and getResourceAsStream method, if multiple resources are found, then the first one must be used.

Since the getResource and getResourceAsStream methods do not support wildcards while the findEntries method does it is necessary to escape the wildcard asterisk ('*' \u002A) with prefixing it with a reverse solidus ('\' \u005C). This implies that a reverse solidus must be escaped with an extra reverse solidus. For example, the path foo\bar* must be escaped to foo\\bar*.

The getResourcePaths method must map to the Bundle getEntryPaths method, its return type is a Set and can not handle multiples. However, the paths from the getEntryPaths method are relative while the methods of the getResourcePaths must be absolute.

For example, assume the following manifest for a bundle:

```
Bundle-ClassPath: localized, WEB-INF
...
```

This WAB has an attached fragment acme-de.jar with the following content:

```
META-INF/MANIFEST.MF
localized/logo.png
```

The getResource method for localized/logo.png uses the findEntries method to find a resource in the directory /localized and the resource logo.png. Assuming the host bundle has no localized/ directory, the Web Runtime must serve the logo.png resource from the acme-de.jar.

128.6.4 Resource Injection and Annotations

The Web Application web.xml descriptor can specify the metadata-complete attribute on the web-app element. This attribute defines whether the web.xml descriptor is *complete*, or whether the classes in the bundle should be examined for deployment annotations. If the metadata-complete attribute is set to true, the Web Runtime must ignore any servlet annotations present in the class files of the Web Application. Otherwise, if the metadata-complete attribute is not specified, or is set to false, the container should process the class files of the Web Application for annotations, if supported.

A WAB can make use of the annotations defined by [7] *JSR 250 Common Annotations for the Java Platform* if supported by the Web Extender. Such a WAB must import the packages the annotations are contained in. A Web Extender that does not support the use of JSR 250 annotations must not process a WAB that imports the annotations package.

128.6.5 Java Server Pages Support

Java Server Pages (JSP) is a rendering technology for template based web page construction. This specification supports [4] *JSP 2.1 specification* if available with the Web Runtime. The servlet element in a web.xml descriptor is used to describe both types of Web Components. JSP components are defined implicitly in the web.xml descriptor through the use of an implicit .jsp extension mapping, or explicitly through the use of a jsp-group element.

128.6.6 Compilation

A Web Runtime compiles a JSP page into a Servlet, either during the deployment phase, or at the time of request processing, and dispatches the request to an instance of such a dynamically created class. Often times, the compilation task is delegated to a separate JSP compiler that will be responsible for identifying the necessary tag libraries, and generating the corresponding Servlet. The container then proceeds to load the dynamically generated class, creates an instance and dispatches the servlet request to that instance.

Supporting in-line compilation of a JSP inside a bundle will require that the Web Runtime maintains a private area where it can store such compiled classes. The Web Runtime can leverage its private bundle storage area. The Web Runtime can construct a special class loader to load generated JSP classes such that classes from the bundle class path are visible to newly compiled JSP classes.

The JSP specification does not describe how JSP pages are dynamically compiled or reloaded. Various Web Runtime implementations handle the aspects in proprietary ways. This specification does not bring forward any explicit requirements for supporting dynamic aspects of JSP pages.

128.7 Security

The security aspects of this specification are defined by the [3] *Servlet 2.5 specification.*

128.8 References

[1] *Java Enterprise Edition Release 5*

https://www.oracle.com/java/technologies/javaee/javaeetechnologies.html#javaee5

[2] *Java EE Web Applications*
https://www.oracle.com/java/technologies/javaee/javaeetechnologies.html#web-app

[3] *Servlet 2.5 specification*
https://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html

[4] *JSP 2.1 specification*
https://jcp.org/aboutJava/communityprocess/final/jsr245/index.html

[5] *PAX Web Extender*
https://ops4j1.jira.com/wiki/spaces/ops4j/pages/3833977/Pax+Web+Extender

[6] *Uniform Resource Locators, RFC 1738*
https://www.ietf.org/rfc/rfc1738.txt

[7] *JSR 250 Common Annotations for the Java Platform*
https://jcp.org/aboutJava/communityprocess/final/jsr250/index.html

130 Coordinator Service Specification

Version 1.0

130.1 Introduction

The OSGi programming model is based on the collaboration of standard and custom components. In such a model there is no central authority that has global knowledge of the complete application. Though this lack of authority can significantly increase reusability (and robustness) there are times when the activities of the collaborators must be coordinated. For example, a service that is repeatedly called in a task could optimize performance by caching intermediate results until it *knew* the task was ended.

To know when a task involving multiple collaborators has ended is the primary purpose of the Coordinator service specification. The Coordinator service provides a rendezvous for an initiator to create a Coordination where collaborators can decide to participate. When the Coordination has ended, all participants are informed.

This Coordinator service provides an explicit Coordination model, the Coordination is explicitly passed as a parameter, and an implicit model where the Coordination is associated with the current thread. Implicit Coordinations can be nested.

Coordinators share the coordination aspects of the resource model of transactions. However, the model is much lighter-weight because it does not support any of the ACID properties.

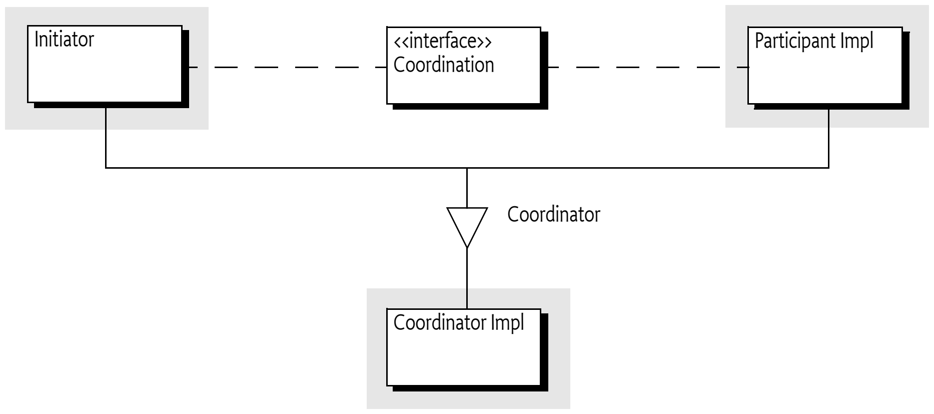
130.1.1 Essentials

- *Coordination* - Provide a solution to allow multiple collaborators to coordinate the outcome of a task initiated by an initiator.
- *Initiator* - An initiator must be able to initiate a coordination and control the final outcome.
- *Participants* - Participants in the task must be informed when the coordination has ended or failed as well as being able to terminate the Coordination.
- *Time-out* - A Coordination should fail after a given time-out.
- *Blocking* - Provide support for blocking and serializing access to Participants.
- *Nesting* - It must be possible to nest Coordinations.
- *Per Thread Model* - Provide a per-thread current Coordination model.
- *Variables* - Provide a variable space per Coordination

130.1.2 Entities

- *Coordinator* - A service that can create and enumerate Coordinations.
- *Coordination* - Represents the ongoing Coordination.
- *Initiator* - The party that initiates a Coordination.
- *Participant* - A party that wants to be informed of the outcome of a Coordination.
- *Collaborator* - Either a participant or initiator.

Figure 130.1 *Class and Service overview*



130.2 Usage

This section is an introduction in the usage of the Coordinator service. It is not the formal specification, the normative part starts at *Coordinator Service* on page 623. This section leaves out some of the details for clarity.

130.2.1 Synopsis

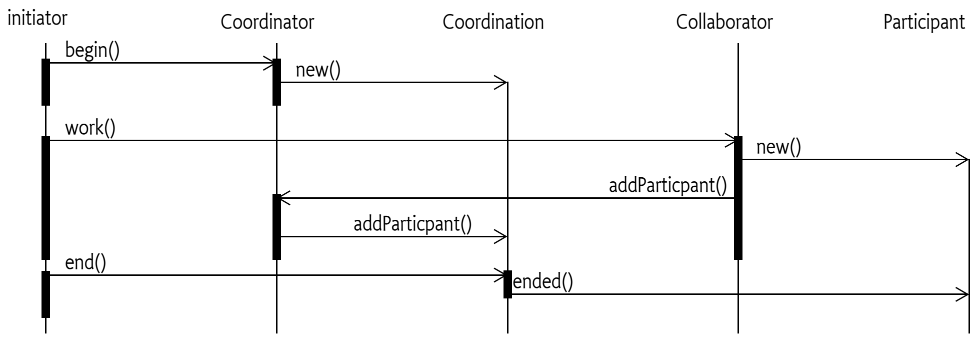
The Coordinator service provides a mechanism for multiple parties to *collaborate* on a common task without a priori knowledge of who will collaborate in that task. A collaborator can participate by adding a *Participant* to the Coordination. The Coordination will notify the Participants when the coordination is *ended* or when it is *failed*.

Each Coordination has an *initiator* that creates the `Coordination` object through the Coordinator service. The initiator can then push this object on a thread-local stack to make it an implicit Coordination or it can pass this object around as a parameter for *explicit* Coordinations. Collaborators can then use the *current* Coordination on the stack or get it from a parameter. Whenever a bundle wants to participate in the Coordination it adds itself to the Coordination as a participant. If necessary, a collaborator can initiate a new Coordination, which could be a nested Coordination for implicit Coordinations.

A Coordination must be *terminated*. Termination is either a normal end when the initiator calls the `end` method or it is failed when the `fail` method is called. A Coordination can be failed by any of the collaborators. A Coordination can also fail independently due to a *time-out* or when the initiator releases its Coordinator service. All participants in the Coordination are informed in reverse participation order about the outcome in a callback for ended or failed Coordinations.

A typical action diagram with a successful outcome is depicted in Figure 130.2.

Figure 130.2 Action Diagram Implicit Coordination



130.2.2 Explicit Coordination

The general pattern for an initiator is to create a Coordination through the Coordinator service, perform the work in a try block, catch any exceptions and fail the Coordination in the catch block, and then ensure ending the Coordination in the finally block. The finally block can cause an exception. This is demonstrated in the following example:

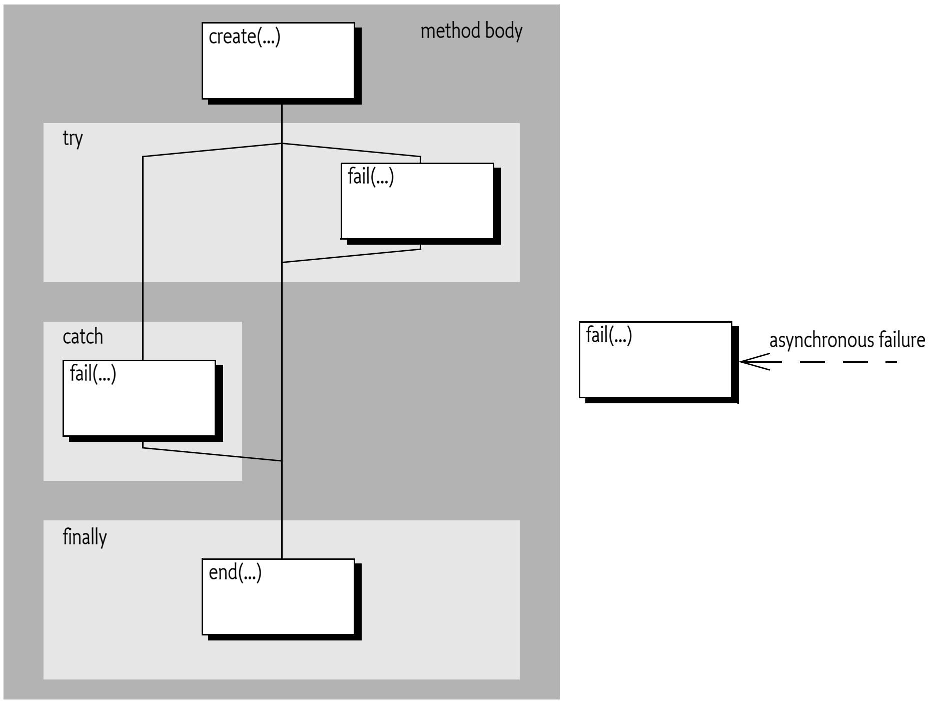
```
Coordination c = coordinator.create("com.example.work",0);
try {
    doWork(c);
} catch( Exception e ) {
    c.fail(e);
} finally {
    c.end();
}
```

This deceptively small template is quite robust:

- If the doWork method throws an Exception then the template fails with a Coordination Exception because it is failed in the try block.
- Any exceptions thrown in the try block are automatically causing the Coordination to fail.
- The Coordination is always terminated and removed from the stack due to the finally block.
- All failure paths, Coordinations that are failed by any of the collaborators, time-outs, or other problems are handled by the end method in the finally block. It will throw a FAILED or PARTIALLY_ENDED Coordination Exception for any of the failures.

The different failure paths and their handling is pictured in Figure 130.3.

Figure 130.3 *Flow through the Coordination template*



The example shows an explicit Coordination because the `create` method is used, implicit Coordinations are used in *Implicit Coordinations* on page 617. The parameters of the `create` method are the name of the Coordination and its time-out. The name is used for informational purposes as well as security. For security reasons, the name must follow the same syntax as the Bundle Symbolic Name. In a secure environment the name can be used to limit Coordinations to a limited set of bundles. For example, a set of bundles signed by a specific signer can use names like `com.acme.*` that are denied to all other bundles.

The zero time-out specifies that the Coordination will not have a time-out. Otherwise it must be a positive long, indicating the number of milliseconds the Coordination may take. However, implementations should have a configurable time-out to ensure that the system remains alive.

In the `doWork` method the real work is done in conjunction with the collaborators. Explicit Coordinations can be passed to other threads if needed. Collaborators can decide to add participants whenever they require a notification when the Coordination has been terminated. For example, the following code could be called from the `doWork` method:

```
void foo(Coordination c) {
  doPrepare();
  c.addParticipant(this);
}
```

This method does the preparation work but does not finalize it so that next time it can use some intermediate results. For example, the `prepare` method could cache a connection to a database that should be reused during the Coordination. The collaborator can assume that it will be called back on either the `failed` or `ended` method. These methods could look like:

```
public void ended(Coordination c)  { doFinish(); }
public void failed(Coordination c) { doFailed(); }
```

The Coordinator provides the guarantee that this code will always call the doFinish method when the Coordination succeeds and doFailed method when it failed.

The Participant must be aware that the ended(Coordination) and failed(Coordination) methods can be called on any thread.

If the doWork method throws an exception it will end up in the catch block of the initiator. The catch block will then fail the Coordination by calling the fail method with the given exception. If the Coordination was already terminated because something else already had failed it then the method call is ignored, only the first fail is used, later fails are ignored.

In all cases, the finally block is executed last. The finally block ends the Coordination. If this coordination was failed then it will throw a Coordination Exception detailing the reason of the failure. Otherwise it will terminate it and notify all the participants.

The Coordination Exception is a Runtime Exception making it unnecessary to declare it.

130.2.3 Multi Threading

Explicit Coordinations allow the Coordination objects to be passed to many different collaborators who can perform the work on different threads. Each collaborator can fail the Coordination at any moment in time or the time-out can occur on yet another thread. Participants must therefore be aware that the callbacks ended and failed can happen on any thread. The following example shows a typical case where a task is parallelized. If any thread fails the Coordination, all other threads could be notified before they're finished.

```
Executor executor = ...
final CountDownLatch latch = new CountdownLatch(10);
final Coordination c = coordinator.create("parallel", 0);
for ( int i=0; i<10; i++) {
  executor.execute(
    new Runnable() {
        public void run() { baz(c); latch.countDown(); }
      });
  }
  latch.await();
  c.end();
```

The Coordination object is thread safe so it can be freely passed around.

130.2.4 Implicit Coordinations

An explicit Coordination requires that the Coordination is passed as a parameter to the doWork method. The Coordinator also supports *implicit* Coordinations. With implicit Coordinations the Coordinator maintains a thread local stack of Coordinations where the top of this stack is the *current* Coordination for that thread. The usage of the implicit Coordination is almost identical to the explicit Coordinations except that all the work occurs on a single thread. The control flow is almost identical to explicit Coordinations:

```
Coordination c = coordinator.begin("com.example.work",0);
try {
    doWork();
} catch( Exception e ) {
    c.fail(e);
} finally {
    c.end();
}
```

See also Figure 130.3. However, in this case the finally block with the call to the end method is even more important. With an implicit Coordination the Coordination is put on a thread local stack in the begin method and must therefore be popped when the Coordination is finished. The finally block ensures therefore the proper cleanup of this thread local stack.

The difference between implicit and explicit Coordinations is that the implicit Coordination is not passed as a parameter, instead, collaborators use the current Coordination. With implicit Coordinations all method invocations in a thread can always access the current Coordination, even if they have many intermediates on the stack. The implicit model allows a collaborator many levels down the stack to detect a current Coordination and register itself without the need to modify all intermediate methods to contain a Coordination parameter. The explicit model has the advantage of explicitness but requires all APIs to be modified to hold the parameter. This model does not support passing the parameter through layers that are not aware of the Coordination. For example, OSGi services in general do not have a Coordination parameter in their methods making the use of explicit Coordinations impossible.

Collaborators can act differently in the presence of a current Coordination. For example, a collaborator can optimize its work flow depending on the presence of a current Coordination.

```
Coordinator coordinator = ...
void foo() {
  doPrepare();
  if ( !coordinator.addParticipant(this))
      doFinish();
}
```

The Coordinator service has an addParticipant method that makes working with the current Coordination simple. If there is a current Coordination then the Coordinator service will add the participant and return true, otherwise it returns false. It is therefore easy to react differently in the presence of a current Coordination. In the previous example, the doFinish method will be called immediately if there was no current Coordination, otherwise it is delayed until the Coordination fails or succeeds. The participant callbacks look the same as in the previous section:

```
public void ended(Coordination c)  { doFinish(); }
public void failed(Coordination c) { doFailed(); }
```

Though the code looks very similar for the implicit and explicit Coordinations there are some additional rules for implicit Coordinations.

The end method must be called on the same thread as the begin method, trying to end it on another thread results in a WRONG_THREAD Coordination Exception being thrown.

Even though the end method must be called on the initiating thread, the callbacks to the Participants can be done on any thread as the specification allows the Coordinator to use multiple threads for all callbacks.

130.2.5 Partial Ending

The Coordination is a best effort mechanism to coordinate, not a transaction model with integrity guarantees. This means that users of the Coordinator service must understand that there are cases where a Coordination ends in limbo. This happens when one of the Participants throws an Exception in the ended callback. This is similar to a transactional resource manager failing to commit in a 2-phase commit after it has voted yes in the prepare phase; a problem that is the cause of much of the complexity of a transaction manager. The Coordinator is limited to use cases that do not require full ACID properties and can therefore be much simpler. However, users of the Coordinator service must be aware of this limitation.

If a Participant throws an exception in the ended method, the end method that terminated the Coordination must throw a PARTIALLY_ENDED Coordination Exception. It is then up to the initiator to correct the situations. In most cases, this means allowing the exception to be re-thrown and handle the failure at the top level. Handling in those cases usually implies logging and continuing.

The following code shows how the PARTIALLY_ENDED case can be handled more explicitly.

```
Coordination c = coordinator.begin("work",0);
try {
  doWork();
} catch( Excption e ) {
  c.fail(e);
} finally {
  try {
    c.end();
  } catch( CoordinationException e ) {
    if ( e.getType() == CoordinationException.PARTIALLY_ENDED) {
      // limbo!
      ...
    }
  }
}
```

130.2.6 Locking

To participate in a Coordination and receive callbacks a collaborator must add a Participant object to the Coordination. The addParticipant(Participant) method blocks if the given Participant object is already used in another Coordination. This blocking facility can be used to implement a number of simple locking schemes that can simplify maintaining state in a concurrent environment.

Using the Participant object as the key for the lock makes it simple to do course grained locking. For example, a service implementation could use the service object as a lock, effectively serializing access to this service when it is used in a Coordination. Coarse grained locking allows all the state to be maintained in the coarse object and not having to worry about multiplexing simultaneous requests. The following code uses the coarse locking pattern because the collaborator implements the Participant interface itself:

```
public class Collaborator implements Participant{
  public void doWork(Coordination coordination ) {
    ...
    coordination.addParticipant(this);
  }

  public void ended(Coordination c) { ... }
  public void failed(Coordination c) { ... }
}
```

The simplicity of the coarse grained locking is at the expense of lower performance because tasks are serialized even if it would have no contention. Locks can therefore also be made more fine grained, allowing more concurrency. In the extreme case, creating a new object for each participation makes it possible to never lock. For example, the following code never locks because it always creates a new object for the Participant:

```
    public void doWork(Coordination coordination){
      final State state = ...
      coordination.addParticipant(
         new Participant() {
           public void ended(Coordination c) { state ... }
           public void failed(Coordination c) { state ...}
    } ); }
```

130.2.7 Failing

Any collaborator can fail an ongoing Coordination by calling the fail(Throwable) method, the Throwable parameter must not be null. When the Coordination has already terminated then this is a no-op. The Coordinator service has a convenience method that fails the current Coordination if present. The fail methods return a boolean that is true when the method call causes the termination of the Coordination, in all other cases it is false.

Failing a Coordination will immediately perform the callbacks and reject any additional Participants by throwing an ALREADY_ENDED Coordination Exception. The asynchronous nature of the fail method implies that it is possible to have been called even before the addParticipant(Participant) method has returned. Anybody that has the Coordination object can check the failed state with the getFailure() method.

In general, the best and most robust strategy to handle failures is to throw an Exception from the collaborator, allowing the initiator to catch the exception and properly fail the Coordination.

130.2.8 Time-out

The time-out is specified in the Coordinator create(String,long) or begin(String,long) methods. A time-out of zero is indefinite, otherwise the time-out specifies the number of milliseconds the Coordination can take to terminate. A given time-out can be extended with the extendTimeout(long) method. This method will add an additional time-out to the existing deadline if a prior deadline was set. For example, the following code extends the time-out with 5 seconds whenever a message must be sent to a remote address:

```
Object sendMessage(Message m) {
  Coordination c = coordinator.peek();
  Address a = m.getDestination();
  if ( c != null && a.isRemote() ) {
    c.extendTimeout(5000);
  }
  return sendMessage0(m);
}
```

Applications should not rely on the exact time-out of the Coordination and only use it as a safety function against deadlocks and hanging collaborators.

130.2.9 Joining

When a Coordination is terminated it is not yet completely finished, the callback to the Participants happens after the atomic termination. In certain cases it is necessary to ensure that a method does not progress until all the participants have been notified. It is therefore possible to wait for the Coordination to completely finish with the join(long) method. This method can have a time-out. For example:

```
void collaborate( final Coordination c ) {
  doWork();
  Thread t = new Thread() {
    public void run(){
      try {
          c.join(0);
          ... // really terminated here, all participantscalled back
      } catch( Exception e) { ... }
    }
  };
  t.start();
}
```

130.2.10 Variables

A Participant is likely to have to maintain state that is particular for the collaboration. This state is usually needed in the ended method to properly finalize the work. In general, the best place to store this state is in the Participant object itself, inner classes and final variables are a good technique for storing the state. However, the state can also be stored in a Coordination *variable*. Each Coordination has a private set of variables that can be obtained with the getVariables() method. The resulting map takes a class as the key and returns an Object. The map is not synchronized, any changes to the map must be synchronized on the returned Map object to ensure the visibility of the changes to other threads. The class used for the key is not related to the returned type, it is a Class object to provide a convenient namespace.

The following example shows how the state can be stored with variables.

```
public void doWork(Coordination coordination){
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    if ( state == null ) {
      state = new State(this);
      map.put( state );
      ... do initial work
    }
  }
  ... do other work
  coordination.addParticipant( this );
}
public void ended(Coordination c) {
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    .. finalize
  }
}
public void failed(Coordination c) {
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    .. finalize
  }
}
```

130.2.11 Optimizing Example

For example, a web based system has a charge service:

```
public interface Charge {
  void charge( String reason, int amount );
}
```

This service is used throughout the system for charging the tasks the system performs. Each servlet request can actually create multiple Charge Data Records (CDR). For this reason, a Coordination is started before the page is constructed. Each part of the page that has an associated cost must create a CDR. There are the following issues at stake:

- Charging should not take place when failing, and
- Performance can be optimized to only persist the CDRs once, and
- The user must be passed to the Charge service.

To begin with the request code:

```
public void doGet(HttpServletRequest rq, HttpServletResponsersp) {
  Coordination c = coordinator.begin("com.acme.request", 30000);
  try {
    Principal p = rq.getUserPrincipal();
    Map<Class<?>,Object> map = c.getVariables();
    map.put( Principal.class, p );
    buildPage(rq,rsp);
  } catch( Exception e  ) { c.fail(e); }
    finally             { c.end(); }
}
```

Each method that has a charge will call the Charge service. The following code shows an implementation of this Charge service.

```
public class ChargeImpl implements Charge,Participant {
  final List<CDR> records = new ArrayList<CDR>();

  public void charge( String reason, int amount ) {
    Coordination c = coordinator.peek();
    if ( c == null ) {
       save( Arrays.asList( new CDR(null, reason, amount)));
    } else {
      Principal p = getPrincipal(c);
      records.add( new CDR(p, reason, amount ) );
      c.addParticipant( this );
    }
  }

  Principal getPrincipal(Coordination c) {
    if ( c == null )
      return null;

    Map<Class<?>,Object> map = c.getVariables();
    synchronized(map) {
      Principal p = (Principal) map.get( Principal.class );
      return p != null ? p : getPrincipal(c.getEnclosingCoordination());
    }
  }

  public void ended(Coordination c) {
    save(records);
    records.clear();
  }
  public void failed(Coordination c) {
    records.clear();
  }

  void save(List<CDR> records) { ... }
}
```

130.2.12 Security Example

The Coordination Permission is a filter based permission that is asserted for many of the methods in the API, the bundle that is checked is always the bundle that created the corresponding Coordination. For example:

```
ALLOW {
     [ BundleSignerCondition "cn=ACME" ]
    ( CoordinationPermission "(signer=cn=ACME)" "*" )
}
```

This example allows bundles signed by ACME to perform all Coordination actions on Coordinations created by bundles signed by ACME.

The filter can also assert the name of the Coordination:

```
coordination.name
```

It is therefore possible to create a name based protection scheme. By denying all bundles except a select group through the use of a name prefix, the use of Coordinations can be restricted to this select group:

```
DENY {
     [ BundleSignerCondition "cn=ACME" "!" ]
    ( CoordinationPermission "(coordination.name=com.acme.*)""*" )
}
ALLOW {
    ( CoordinationPermission "(coordination.name=*)" "*" )
}
```

If a bundle is not signed by ACME it will be denied the use of Coordination names starting with `com.acme.` though it will be allowed to use any other name. This effectively enables only bundles signed by ACME to create Coordinations with this name prefix.

130.3 Coordinator Service

The Coordinator service is the entry point for the Coordination. It provides the following functions:

- Coordination creation
- Life cycle management of a Coordination
- Thread based Coordinations
- Introspection

130.3.1 Coordination Creation

A Coordination object is created by an *initiator*. An initiator can create a Coordination object with the Coordinator `create(String,long)` or `begin(String,long)` method. Each Coordination when created gets a positive long identity that is available with `getId()`. Ids are a unique identifier for a specific Coordinator service. The id is always increasing, that is, a Coordination with a higher id is created later.

The create methods specify the name of the Coordination. This name is a security concept, see *Security* on page 628, as well as used for debugging. The coordination name must therefore conform to the same syntax as a bundle symbolic name:

```
coordination-name ::= symbolic-name    // see OSGi Core Release 8
```

Passing a name that does not conform to this syntax must throw an Illegal Argument Exception. There are no constraints on duplicates, multiple different Coordinations can use the same name. The name of the Coordination is available with the getName() method.

130.3.2 Adding Participants

The Coordination object can be passed to *collaborators* as a parameter in a method call. Some of these collaborators might be interested in *participating* in the given Coordination, they can achieve this by adding a Participant object to the Coordination.

A Participant is a collaborator that requires a callback after the Coordination has been terminated, either when it ended or when it failed. To participate, it must add a Participant object to a Coordination with the addParticipant(Participant) method on Coordination. This method throws an ALREADY_ENDED or FAILED Coordination Exception when the Coordination has been terminated.

When a Participant is:

- *Not in any Coordination* - Add it to the given Coordination and return.
- *In target Coordination* - Ignore, participant is already present. A Participant can participate in the same Coordination multiple times by calling addParticipant(Participant) but will only be called back once when the Coordination is terminated. Its order must be defined by the first addition.
- *In another Coordination* - Lock until after the other Coordination has notified all the Participants. Implementations can detect deadlocks in certain cases and throw a Coordination Exception if a dead lock exist, otherwise the deadlock is solved when the Coordination times out.

Verifying if a Participant object is already in another Coordination must use identity and not equality.

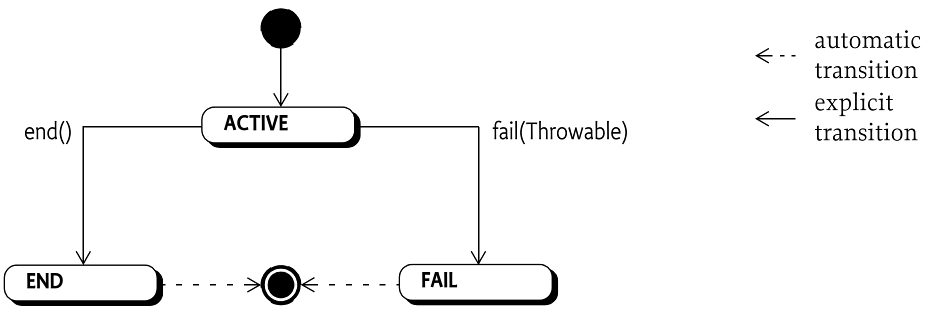
130.3.3 Active

A Coordination is active until it is *terminated*. A Coordination can terminate because it is *ended*, or it is *failed*. The following methods cause a termination:

- end() - A normal end. All participants that were added before the end call are called back on their ended(Coordination) method.
- fail(Throwable) - The Coordination has failed, this will call back the failed(Coordination) method on the participants. This method can be called by the Coordinator, the initiator, or any of the collaborators. There are a number of failures that are built in to the Coordinator. These failures use singleton Exception instances defined in the Coordination interface:
  - TIMEOUT - If the Coordination times out the Coordination is failed with the TIMEOUT exception instance in Coordination.
  - RELEASED - If the Coordinator that created the Coordination was unget, all Coordinations created by it will fail with the RELEASED exception.

The state diagram for the Coordination is pictured in Figure 130.4.

Figure 130.4 *Coordination state diagram*



130.3.4 Explicit and Implicit Models

The Coordinator supports two very different models of usage: *explicit* and *implicit*. The explicit model is when a Coordination is created and passed around as a parameter. The second model is the implicit model where the Coordinator maintains a thread local stack of Coordinations. Any collaborator can then decide to use the top of the stack as the *current* Coordination. The peek() method provides access to the current Coordination.

The begin(String,long) method creates a new Coordination and pushes this on the stack, beginning an implicit Coordination. This is identical to:

```
coordinator.create("work",0).push();
```

Once a Coordination is pushed on a stack it is from that moment on associated with the current thread. A Coordination can only be pushed once, the ALREADY_PUSHED Coordination Exception must be thrown when the Coordination is already associated with one of the thread local stacks maintained by the Coordinator service.

The Coordination is removed from the stack in the end() method. The end() method must not only terminate itself but it must also terminate all nested Coordinations.

The current Coordination can also be explicitly removed with the Coordinator pop() method.

A Coordination that is pushed on a thread local stack returns the associated thread on the getThread() method. This method returns null for Coordinations not on any stack, that is, explicit Coordinations.

130.3.5 Termination

Both the end() and fail(Throwable) methods terminate the Coordination if it was not already terminated. Termination is atomic, only the end or the fail method can terminate the Coordination. Though this happens on different threads, a Coordination can never both end and fail from any perspective. That is, if a fail races with end then only one of them can win and the other provides the feedback that the Coordination was already terminated.

Terminating a Coordination has the following effects:

- It is atomic, it can only happen once in a Coordination
- It freezes the set of participants, no more participants can be added

130.3.6 Ending

The end() method should always be called at the end of a Coordination to ensure proper termination, notification, and cleanup. The end method throws a FAILED or PARTIALLY_ENDED Coordination Exception if the Coordination was failed before.

If the Coordination had already been ended before then this is a programming error and an ALREADY_ENDED Configuration Exception is thrown. The end() method should never be called twice on the same Coordination.

If the termination succeeds then the participants must be notified by calling the ended(Coordination) method on each Participant that had been successfully added to the Coordination. This callback can take place on any thread but must be in reverse order of adding. That is, the last added Participant is called back first.

Participants must never make any assumptions about the current Coordination in the callback. The Coordination it was added to is therefore given as an explicit parameter in the ended(Coordination) method.

If a Participant throws an Exception then this must not prevent the calling of the remaining participants. The Exception should be logged. If a Participant has thrown an Exception then the end()

method must throw a PARTIALLY_ENDED Coordination Exception after the last Participant has returned from its callback, otherwise the method returns normally. Participants should normally not throw Exceptions in their callbacks.

If the Coordination is implicit (it is pushed on a stack) then the Coordination must be removed from its stack after the participants have been called back. This requires that the ending thread is the same as the thread of the Coordination. The end thread is the thread of the end() method call. If the Coordination's thread is not the same as the ending thread then a WRONG_THREAD Coordination Exception is thrown.

If the ending Coordination is on the stack but it is not the current Coordination then each nested Coordination must be ended before the current Coordination, see *Nesting Implicit Coordinations* on page 626 for more information.

The fail(Throwable) method must not remove the current Coordination, it must remain on the stack. The initiator must always call the end() method. Always calling end() in a finally block is therefore paramount.

130.3.7 Failing, TIMEOUT, ORPHANED, and RELEASED

Failing can happen asynchronously during the time a Coordination is active. A Coordination is failed by calling fail(Throwable). The Throwable argument must not be null, it is the cause of the failure.

Failing a Coordination must first terminate it. If the Coordination was already terminated the fail(Throwable) method has no effect. Otherwise, it must callback all its added Participants on the failed(Coordination) callback method. Exceptions thrown from this method should be logged and further ignored. The callback can occur on any thread, including the caller's.

Implicit Coordinations must not be popped from its stack in a fail nor is it necessary to call the fail method from any particular thread. The removal of the Coordination from the stack must happen in the end method.

There are two asynchronous events that can also fail the Coordination. If the Coordination times out, it will be treated as a fail(TIMEOUT) and if the Coordinator is ungotten with active Coordinations then each of those Coordinations must fail as if fail(RELEASED) is called.

A Coordination can also be *orphaned*. An orphaned Coordination has no longer any outside references. This means that the Coordination can no longer be ended or failed. Such Coordinations must fail with an ORPHANED Exception.

130.3.8 Nesting Implicit Coordinations

Implicit Coordinations can be nested. For this reason, the Coordinator maintains a thread local stack of Coordinations where the top, accessible with the peek() method, is the current Coordination. Each time a new Coordination is begun with the begin(String,long) method, the current Coordination is replaced with the newly created Coordination. When that Coordination is ended, the previous current Coordination is restored. Nesting is always on the same thread, implicit Coordinations are always associated with a single thread, available through its getThread() method. The end method must be called on the same thread as the begin(String,long) or last push() method.

Using the standard model for implicit Coordinations, where the initiator always ends the Coordination on the same thread as it begun, ensures that nesting is properly handled. However, in certain cases it is necessary to manipulate the stack or make implicit Coordinations explicit or vice versa. For this reason, it is possible to pop Coordinations from the stack with the pop() method. This method disassociates the Coordination from the current thread and restores the previous (if any) Coordination as the current Thread. A Coordination can then be made the current Coordination for a thread by calling the push() method. However, a Coordination can be pushed on the stack at most once. If a Coordination is pushed a second time, in any thread, the ALREADY_PUSHED Coordination Exception must be thrown.

The Coordination is removed from its stack when the end() method is called. It is therefore highly recommended to always end a Coordination in the nesting order. However, it is possible that a Coordination is ended that is not the current Coordination, it has nested Coordinations that were not properly ended. In that case all nested Coordinations must be ended in reverse creation order, that is, the current Coordination first, by calling the end method on it.

If any Coordination fails to end properly (including PARTIALLY_ENDED) then the remaining Coordinations on the stack must fail and chain the exceptions. In pseudo code:

```
while (coordinator.peek() != this) {
 try {
     coordinator.peek().end();
 } catch (CoordinationException e) {
      coordinator.peek().fail(e);
 }
}
```

130.3.9 Time-outs

When a Coordination is created it will receive a time-out. A time-out is a positive value or zero. A zero value indicates that the Coordination should have no time-out. This does not imply that a Coordination will never time-out, implementations are allowed to be configured with a limit to the maximum active time for a Coordination.

Collaborators can extend the time out with the extendTimeout(long) method. If no time-out was set (0), this method will be ignored. Otherwise the given amount (which must be positive) is added to the existing deadline. A Coordinator implementation can fail the Coordination earlier, however, when configured to do so.

If a Coordination is timed out, the Coordination is failed with a fail(TIMEOUT) method call from an unspecified thread, see *Failing, TIMEOUT, ORPHANED, and RELEASED* on page 626.

130.3.10 Released

The Coordination's life cycle is bound to the Coordinator service that created it. If the initiator's bundle ungets this service then the Coordinator must fail all the Coordinations created by this Coordinator by calling the fail(RELEASED) method.

Participants from bundles that are stopped are not taken into account. This means that it is possible that a participant is called while its bundle is stopped. Stopped Participants should fail any Coordinations that they participate in.

130.3.11 Coordinator Convenience Methods

The Coordinator contains a number of convenience methods that can be used by collaborators to interact with the current Coordination.

- begin(String,long) - Is logically the same as create(String,long). push().
- addParticipant(Participant) - This method makes it easy to react differently to the presence of a current implicit Coordination. If a current Coordination exists, the participant is added and true is returned (or an exception thrown if the Coordination is already terminated), otherwise false is returned.
- fail(Throwable) - If there is no current Coordination, this method returns false. Otherwise it returns the result of calling fail(Throwable) on the current Coordination. This method therefore only returns true when a current Coordination was actually terminated due to this call.

130.3.12 Administrative Access

The Coordination objects provide a number of methods that are used for administrating the Coordinations and the Coordinator.

- getBundle() - Provide the bundle that created the Coordination. This bundle is the bundle belonging to the Bundle Context used to get the Coordinator service.
- getFailure() - The Exception that caused this Coordination to fail or null. There are two fixed exception instances for a time out (TIMEOUT), when the Coordination is orphaned (ORPHANED), and when the Coordinator service is released (RELEASED).
- getId() - The Coordination's id.
- getName() - The name of the Coordination.
- getParticipants() - The current list of participants. This is a mutable snapshot of the added participants. Changing the snapshot has no effect on the Coordination.
- getThread() - Answer the thread associated with an implicit Coordination. If the Coordination is not implicit then the answer is null.
- getEnclosingCoordination() - Return the enclosing Coordination.

And for the Coordinator:

- getCoordination(long) - Retrieve a Coordination by its id.
- getCoordinations() - Get a list of active Coordinations

130.3.13 Summary

A Coordination can exist in three different states *ACTIVE*, *END*, and *FAIL*. During its life it will transition from ACTIVE to either END or FAIL. The entry (when the state is entered) and exit (when the state is left) actions when this transition takes place and the effect on the different methods are summarized in the following table.

Table 130.1 *States and transitions*

State/Method	ACTIVE	END	FAIL
entry action		Notify all the participants by calling the ended(Coordination) method.	Notify all the participants by calling the failed(Coordination) method.
exit action	Terminate		
end()	-> END. Can throw PARTIALLY_ENDED	throws ALREADY_ENDED	throws FAILED
fail(Throwable)	-> FAIL, return true.	return false.	return false.

130.4 Security

This specification provides a Coordination Permission. This permission can enforce the name of the coordination as well as assert the properties of the initiating bundle, like for example the signer or bundle symbolic name. The permission therefore uses a filter as name, as defined in the filter based permissions section in *OSGi Core Release 8*, see *OSGi Core Release 8*. There is one additional parameter for the filter:

```
coordination.name
```

The value is the given name of the Coordination. Restricting the name of a Coordination allows the deployer to limit the use of this name to a restricted set of bundles.

The following actions are defined:

- INITIATE - Required to initiate and control a Coordination.
- PARTICIPATE - Required to participate in a Coordination.

- ADMIN - Required to administrate a Coordinator.

The target bundle of the Coordination Permission is the initiator's bundle. This is the bundle that got the Coordinator service to create the Coordination. An initiator must therefore have permission to create Coordinations for itself.

There are two constructors available:

- CoordinationPermission(String,String) - The constructor for the granted permission. It is given a filter expression and the actions that the permission applies to.
- CoordinationPermission(String,Bundle,String) - The constructor for the requested permission. It is given the name of the permission, the bundle that created the corresponding coordination, and the requested actions.

130.5 org.osgi.service.coordinator

Coordinator Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.coordinator; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.coordinator; version="[1.0,1.1)"
```

130.5.1 Summary

- Coordination - A Coordination object is used to coordinate a number of independent Participants.
- CoordinationException - Unchecked exception which may be thrown by a Coordinator implementation.
- CoordinationPermission - A bundle's authority to create or use a Coordination.
- Coordinator - A Coordinator service coordinates activities between different parties.
- Participant - A Participant participates in a Coordination.

130.5.2 public interface Coordination

A Coordination object is used to coordinate a number of independent Participants.

Once a Coordination is created, it can be used to add Participant objects. When the Coordination is ended, the participants are notified. A Coordination can also fail for various reasons. When this occurs, the participants are notified of the failure.

A Coordination must be in one of two states, either ACTIVE or TERMINATED. The transition between ACTIVE and TERMINATED must be atomic, ensuring that a Participant can be guaranteed of either receiving an exception when adding itself to a Coordination or of receiving notification the Coordination has terminated.

A Coordination object is thread safe and can be passed as a parameter to other parties regardless of the threads these parties use.

The following example code shows how a Coordination should be used.

```
 void foo() {
   Coordination c = coordinator.create("work", 0);
   try {
     doWork(c);
   }
   catch (Exception e) {
     c.fail(e);
   }
   finally {
     c.end();
   }
 }
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

130.5.2.1 public static final Exception ORPHANED

A singleton exception that will be the failure cause when a Coordination has been orphaned.

130.5.2.2 public static final Exception RELEASED

A singleton exception that will be the failure cause when the Coordinations created by a bundle are terminated because the bundle released the Coordinator service.

130.5.2.3 public static final Exception TIMEOUT

A singleton exception that will be the failure cause when a Coordination times out.

130.5.2.4 public void addParticipant(Participant participant)

participant The Participant to register with this Coordination. The participant must not be `null`.

□ Register a Participant with this Coordination.

Once a Participant is registered with this Coordination, it is guaranteed to receive a notification for either normal or failure termination when this Coordination is terminated.

Participants are registered using their object identity. Once a Participant is registered with this Coordination, subsequent attempts to register the Participant again with this Coordination are ignored and the Participant is only notified once when this Coordination is terminated.

A Participant can only be registered with a single active Coordination at a time. If a Participant is already registered with an active Coordination, attempts to register the Participation with another active Coordination will block until the Coordination the Participant is registered with terminates. Notice that in edge cases the notification to the Participant that this Coordination has terminated can happen before this method returns.

Attempting to register a Participant with a terminated Coordination will result in a CoordinationException being thrown.

The ordering of notifying Participants must follow the reverse order in which the Participants were registered.

Throws `CoordinationException`– If the Participant could not be registered with this Coordination. This exception should normally not be caught by the caller but allowed to be caught by the initiator of this Coordination.

`SecurityException`– If the caller does not have `CoordinationPermission[PARTICIPATE]` for this Coordination.

130.5.2.5 public void end()

□ Terminate this Coordination normally.

If this Coordination has been pushed on the thread local Coordination stack of another thread, this method does nothing except throw a CoordinationException of type CoordinationException.WRONG_THREAD.

If this Coordination has been pushed on the thread local Coordination stack of this thread but is not the current Coordination, then the Coordinations on the thread local Coordination stack above this Coordination must be terminated and removed from the thread local Coordination stack before this Coordination is terminated. Each of these Coordinations, starting with the current Coordination, will be terminated normally . If the termination throws a CoordinationException, then the next Coordination on the thread local Coordination stack will be terminated as a failure with a failure cause of the thrown CoordinationException. At the end of this process, this Coordination will be the current Coordination and will have been terminated as a failure if any of the terminated Coordinations threw a CoordinationException

If this Coordination is the current Coordination, then it will be removed from the thread local Coordination stack.

If this Coordination is already terminated, a CoordinationException is thrown. If this Coordination was terminated as a failure, the failure cause will be the cause of the thrown CoordinationException.

Otherwise, this Coordination is terminated normally and then all registered Participants are notified. Participants should finalize any work associated with this Coordination. The successful return of this method indicates that the Coordination has terminated normally and all registered Participants have been notified of the normal termination.

It is possible that one of the Participants throws an exception during notification. If this happens, this Coordination is considered to have partially failed and this method must throw a CoordinationException of type CoordinationException.PARTIALLY_ENDED after all the registered Participants have been notified.

Throws `CoordinationException`– If this Coordination has failed, including timed out, or partially failed or this Coordination is on the thread local Coordination stack of another thread.

`SecurityException`– If the caller does not have `CoordinationPermission[INITIATE]` for this Coordination.

130.5.2.6 public long extendTimeout(long timeMillis)

timeMillis The time in milliseconds to extend the current timeout. If the initial timeout was specified as 0, no extension must take place. A zero must have no effect.

□ Extend the time out of this Coordination.

Participants can call this method to extend the timeout of this Coordination with at least the specified time. This can be done by Participants when they know a task will take more than normal time.

This method will return the new deadline if an extension took place or the current deadline if, for whatever reason, no extension takes place. Note that if a maximum timeout is in effect, the deadline may not be extended by as much as was requested, if at all. If there is no deadline, zero is returned. Specifying a timeout extension of 0 will return the existing deadline.

Returns The new deadline in milliseconds. If the specified time is 0, the existing deadline is returned. If this Coordination was created with an initial timeout of 0, no timeout is set and 0 is returned.

Throws `CoordinationException`– If this Coordination is terminated.

`IllegalArgumentException`– If the specified time is negative.

`SecurityException`– If the caller does not have `CoordinationPermission[PARTICIPATE]` for this Coordination.

130.5.2.7 **public boolean fail(Throwable cause)**

cause The failure cause. The failure cause must not be null.

□ Terminate this Coordination as a failure with the specified failure cause.

If this Coordination is already terminated, this method does nothing and returns false.

Otherwise, this Coordination is terminated as a failure with the specified failure cause and then all registered Participants are notified. Participants should discard any work associated with this Coordination. This method will return true.

If this Coordination has been pushed onto a thread local Coordination stack, this Coordination is not removed from the stack. The creator of this Coordination must still call end() on this Coordination to cause it to be removed from the thread local Coordination stack.

Returns true if this Coordination was active and was terminated by this method, otherwise false.

Throws SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.8 **public Bundle getBundle()**

□ Returns the bundle that created this Coordination. This is the bundle that obtained the Coordinator service that was used to create this Coordination.

Returns The bundle that created this Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.9 **public Coordination getEnclosingCoordination()**

□ Returns the Coordination enclosing this Coordination if this Coordination is on the thread local Coordination stack.

When a Coordination is pushed onto the thread local Coordination stack, the former current Coordination, if any, is the enclosing Coordination of this Coordination. When this Coordination is removed from the thread local Coordination stack, this Coordination no longer has an enclosing Coordination.

Returns The Coordination enclosing this Coordination if this Coordination is on the thread local Coordination stack or null if this Coordination is not on the thread local Coordination stack or has no enclosing Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.10 **public Throwable getFailure()**

□ Returns the failure cause of this Coordination.

If this Coordination has failed, then this method will return the failure cause.

If this Coordination timed out, this method will return TIMEOUT as the failure cause. If this Coordination was active when the bundle that created it released the Coordinator service, this method will return RELEASED as the failure cause. If the Coordination was orphaned, this method will return ORPHANED as the failure cause.

Returns The failure cause of this Coordination or null if this Coordination has not terminated as a failure.

Throws SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.2.11 public long getId()

□ Returns the id assigned to this Coordination. The id is assigned by the Coordinator service which created this Coordination and is unique among all the Coordinations created by the Coordinator service and must not be reused as long as the Coordinator service remains registered. The id must be positive and monotonically increases for each Coordination created by the Coordinator service.

Returns The id assigned to this Coordination.

130.5.2.12 public String getName()

□ Returns the name of this Coordination. The name is specified when this Coordination was created.

Returns The name of this Coordination.

130.5.2.13 public List<Participant> getParticipants()

□ Returns a snapshot of the Participants registered with this Coordination.

Returns A snapshot of the Participants registered with this Coordination. If no Participants are registered with this Coordination, the returned list will be empty. The list is ordered in the order the Participants were registered. The returned list is the property of the caller and can be modified by the caller.

Throws SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.2.14 public Thread getThread()

□ Returns the thread in whose thread local Coordination stack this Coordination has been pushed.

Returns The thread in whose thread local Coordination stack this Coordination has been pushed or null if this Coordination is not in any thread local Coordination stack.

Throws SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.15 public Map<Class<?>, Object> getVariables()

□ Returns the variable map associated with this Coordination. Each Coordination has a map that can be used for communicating between different Participants. The key of the map is a class, allowing for private data to be stored in the map by using implementation classes or shared data by using shared interfaces. The returned map is not synchronized. Users of the map must synchronize on the Map object while making changes.

Returns The variable map associated with this Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[PARTICIPANT] for this Coordination.

130.5.2.16 public boolean isTerminated()

□ Returns whether this Coordination is terminated.

Returns true if this Coordination is terminated, otherwise false if this Coordination is active.

130.5.2.17 public void join(long timeMillis) throws InterruptedException

timeMillis Maximum time in milliseconds to wait. Specifying a time of 0 will wait until this Coordination is terminated.

□ Wait until this Coordination is terminated and all registered Participants have been notified.

Throws InterruptedException– If the wait is interrupted.

IllegalArgumentException– If the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.18 public Coordination push()

□ Push this Coordination object onto the thread local Coordination stack to make it the current Coordination.

Returns This Coordination.

Throws CoordinationException– If this Coordination is already on the any thread's thread local Coordination stack or this Coordination is terminated.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.3 public class CoordinationException extends RuntimeException

Unchecked exception which may be thrown by a Coordinator implementation.

130.5.3.1 public static final int ALREADY_ENDED = 4

The Coordination has already terminated normally.

130.5.3.2 public static final int ALREADY_PUSHED = 5

The Coordination was already on a thread's thread local Coordination stack.

130.5.3.3 public static final int DEADLOCK_DETECTED = 1

Registering a Participant with a Coordination would have resulted in a deadlock.

130.5.3.4 public static final int FAILED = 2

The Coordination has terminated as a failure with Coordination.fail(Throwable). When this exception type is used, the getCause() method must return a non-null value.

130.5.3.5 public static final int LOCK_INTERRUPTED = 6

The current thread was interrupted while waiting to register a Participant with a Coordination.

130.5.3.6 public static final int PARTIALLY_ENDED = 3

The Coordination has partially ended.

130.5.3.7 public static final int UNKNOWN = 0

Unknown reason for this exception.

130.5.3.8 public static final int WRONG_THREAD = 7

The Coordination cannot be ended by the calling thread since the Coordination is on the thread local Coordination stack of another thread.

130.5.3.9 public CoordinationException(String message, Coordination coordination, int type, Throwable cause)

message The detail message for this exception.

coordination The Coordination associated with this exception.

cause The cause associated with this exception.

type The type of this exception.

□ Create a new Coordination Exception with a cause.

Throws IllegalArgumentException– If the specified type is FAILED and the specified cause is null.

130.5.3.10 public CoordinationException(String message, Coordination coordination, int type)

message The detail message for this exception.

coordination The Coordination associated with this exception.

type The type of this exception.

□ Create a new Coordination Exception.

Throws IllegalArgumentException– If the specified type is FAILED .

130.5.3.11 public long getId()

□ Returns the id of the Coordination associated with this exception.

Returns The id of the Coordination associated with this exception or -1 if no Coordination is associated with this exception.

130.5.3.12 public String getName()

□ Returns the name of the Coordination associated with this exception.

Returns The name of the Coordination associated with this exception or "<>" if no Coordination is associated with this exception.

130.5.3.13 public int getType()

□ Returns the type for this exception.

Returns The type of this exception.

130.5.4 public final class CoordinationPermission extends BasicPermission

A bundle's authority to create or use a Coordination.

CoordinationPermission has three actions: initiate, participate and admin.

Concurrency Thread-safe

130.5.4.1 public static final String ADMIN = "admin"

The action string admin.

130.5.4.2 public static final String INITIATE = "initiate"

The action string initiate.

130.5.4.3 public static final String PARTICIPATE = "participate"

The action string participate.

130.5.4.4 public CoordinationPermission(String filter, String actions)

filter A filter expression. Filter attribute names are processed in a case sensitive manner. A special value of "*" can be used to match all coordinations.

actions admin, initiate or participate (canonical order).

□ Creates a new granted CoordinationPermission object. This constructor must only be used to create a permission that is going to be checked.

Examples:

```
(coordination.name=com.acme.*)
(&(signer=\*,o=ACME,c=US)(coordination.name=com.acme.*))
(signer=\*,o=ACME,c=US)
```

When a signer key is used within the filter expression the signer value must escape the special filter chars ('*', '(', ')').

The name is specified as a filter expression. The filter gives access to the following attributes:

- signer - A Distinguished Name chain used to sign the exporting bundle. Wildcards in a DN are not matched according to the filter string rules, but according to the rules defined for a DN chain.
- location - The location of the exporting bundle.
- id - The bundle ID of the exporting bundle.
- name - The symbolic name of the exporting bundle.
- coordination.name - The name of the requested coordination.

Filter attribute names are processed in a case sensitive manner.

Throws IllegalArgumentException– If the filter has an invalid syntax.

130.5.4.5 public CoordinationPermission(String coordinationName, Bundle coordinationBundle, String actions)

coordinationName The name of the requested Coordination.

coordinationBundle The bundle which created the requested Coordination.

actions admin, initiate or participate (canonical order).

□ Creates a new requested CoordinationPermission object to be used by the code that must perform checkPermission. CoordinationPermission objects created with this constructor cannot be added to an CoordinationPermission permission collection.

130.5.4.6 public boolean equals(Object obj)

obj The object to test for equality with this CoordinationPermission object.

□ Determines the equality of two CoordinationPermission objects. This method checks that specified permission has the same name and CoordinationPermission actions as this CoordinationPermission object.

Returns true if obj is a CoordinationPermission, and has the same name and actions as this CoordinationPermission object; false otherwise.

130.5.4.7 public String getActions()

□ Returns the canonical string representation of the CoordinationPermission actions.

Always returns present CoordinationPermission actions in the following order: admin, initiate, participate.

Returns Canonical string representation of the CoordinationPermission actions.

130.5.4.8 public int hashCode()

□ Returns the hash code value for this object.

Returns A hash code value for this object.

130.5.4.9 public boolean implies(Permission p)

p The requested permission.

□ Determines if the specified permission is implied by this object.

This method checks that the filter of the target is implied by the coordination name of this object. The list of CoordinationPermission actions must either match or allow for the list of the target object to imply the target CoordinationPermission action.

Returns true if the specified permission is implied by this object; false otherwise.

130.5.4.10 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing CoordinationPermission objects.

Returns A new PermissionCollection object.

130.5.5 public interface Coordinator

A Coordinator service coordinates activities between different parties.

A bundle can use the Coordinator service to create Coordination objects. Once a Coordination object is created, it can be pushed on the thread local Coordination stack to be an implicit parameter as the current Coordination for calls to other parties, or it can be passed directly to other parties as an argument. The current Coordination, which is on the top of the current thread's thread local Coordination stack, can be obtained with peek().

Any active Coordinations created by a bundle must be terminated when the bundle releases the Coordinator service. The Coordinator service must fail these Coordinations with the RELEASED exception.

A Participant can register to participate in a Coordination and receive notification of the termination of the Coordination.

The following example code shows a example usage of the Coordinator service.

```
void foo() {
  Coordination c = coordinator.begin("work", 0);
  try {
    doWork();
  } catch (Exception e) {
    c.fail(e);
  } finally {
    c.end();
  }
}
```

In the doWork method, code can be called that requires notification of the termination of the Coordination. The doWork method can then register a Participant with the Coordination.

```
void doWork() {
  if (coordinator.addParticipant(this)) {
    beginWork();
  } else {
    beginWork();
    finishWork();
  }
}

void ended(Coordination c) {
  finishWork();
}

void failed(Coordination c) {
  undoWork();
}
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

130.5.5.1 public boolean addParticipant(Participant participant)

participant The Participant to register with the current Coordination. The participant must not be null.

□ Register a Participant with the current Coordination.

If there is no current Coordination, this method does nothing and returns false.

Otherwise, this method calls Coordination.addParticipant(Participant) with the specified Participant on the current Coordination and returns true.

Returns false if there was no current Coordination, otherwise returns true.

Throws CoordinationException– If the Participant could not be registered with the current Coordination. This exception should normally not be caught by the caller but allowed to be caught by the initiator of this Coordination.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for the current Coordination.

See Also Coordination.addParticipant(Participant)

130.5.5.2 public Coordination begin(String name, long timeMillis)

name The name of this coordination. The name does not have to be unique but must follow the symbolic-name syntax from the Core specification.

timeMillis Timeout in milliseconds. A value of 0 means no timeout is required. If the Coordination is not terminated within the timeout, the Coordinator service will fail the Coordination with a TIMEOUT exception.

□ Create a new Coordination and make it the current Coordination.

This method does that same thing as calling create(name, timeMillis).push()

Returns A new Coordination object

Throws IllegalArgumentException– If the specified name does not follow the symbolic-name syntax or the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the specified name and creating bundle.

130.5.5.3 public Coordination create(String name, long timeMillis)

name The name of this coordination. The name does not have to be unique but must follow the symbolic-name syntax from the Core specification.

timeMillis Timeout in milliseconds. A value of 0 means no timeout is required. If the Coordination is not terminated within the timeout, the Coordinator service will fail the Coordination with a TIMEOUT exception.

□ Create a new Coordination.

Returns The new Coordination object.

Throws IllegalArgumentException– If the specified name does not follow the symbolic-name syntax or the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the specified name and creating bundle.

130.5.5.4 public boolean fail(Throwable cause)

cause The failure cause. The failure cause must not be null .

□ Terminate the current Coordination as a failure with the specified failure cause.

If there is no current Coordination, this method does nothing and returns false.

Otherwise, this method returns the result from calling Coordination.fail(Throwable) with the specified failure cause on the current Coordination.

Returns false if there was no current Coordination, otherwise returns the result from calling Coordination.fail(Throwable) on the current Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for the current Coordination.

See Also Coordination.fail(Throwable)

130.5.5.5 public Coordination getCoordination(long id)

id The id of the requested Coordination.

□ Returns the Coordination with the specified id.

Returns A Coordination having with specified id or null if no Coordination with the specified id exists, the Coordination with the specified id is terminated or the caller does not have CoordinationPermission[ADMIN] for the Coordination with the specified id.

130.5.5.6 public Collection<Coordination> getCoordinations()

□ Returns a snapshot of all active Coordinations.

Since Coordinations can be terminated at any time, Coordinations in the returned collection can be terminated before the caller examines the returned collection.

The returned collection must only contain the Coordinations for which the caller has CoordinationPermission[ADMIN].

Returns A snapshot of all active Coordinations. If there are no active Coordinations, the returned list will be empty. The returned collection is the property of the caller and can be modified by the caller.

130.5.5.7 public Coordination peek()

□ Returns the current Coordination.

The current Coordination is the Coordination at the top of the thread local Coordination stack. If the thread local Coordination stack is empty, there is no current Coordination. Each Coordinator service maintains thread local Coordination stacks.

This method does not alter the thread local Coordination stack.

Returns The current Coordination or null if the thread local Coordination stack is empty.

130.5.5.8 public Coordination pop()

□ Remove the current Coordination from the thread local Coordination stack.

The current Coordination is the Coordination at the top of the thread local Coordination stack. If the thread local Coordination stack is empty, there is no current Coordination. Each Coordinator service maintains its own thread local Coordination stacks.

This method alters the thread local Coordination stack, if it is not empty, by removing the Coordination at the top of the thread local Coordination stack.

Returns The Coordination that was the current Coordination or null if the thread local Coordination stack is empty.

Throws SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the current Coordination.

130.5.6 public interface Participant

A Participant participates in a Coordination.

A Participant can participate in a Coordination by registering itself with the Coordination. After successfully registering itself, the Participant is notified when the Coordination is terminated.

If a Coordination terminates normally, then all registered Participants are notified on their ended(Coordination) method. If the Coordination terminates as a failure, then all registered Participants are notified on their failed(Coordination) method.

Participants are required to be thread safe as notification can be made on any thread.

A Participant can only be registered with a single active Coordination at a time. If a Participant is already registered with an active Coordination, attempts to register the Participation with another active Coordination will block until the Coordination the Participant is registered with terminates. Notice that in edge cases the notification to the Participant that the Coordination has terminated can happen before the registration method returns.

Concurrency Thread-safe

130.5.6.1 public void ended(Coordination coordination) throws Exception

coordination The Coordination that has terminated normally.

□ Notification that a Coordination has terminated normally.

This Participant should finalize any work associated with the specified Coordination.

Throws `Exception`– If this Participant throws an exception, the Coordinator service should log the exception. The Coordination.end() method which is notifying this Participant must continue notification of other registered Participants. When this is completed, the Coordination.end() method must throw a CoordinationException of type CoordinationException.PARTIALLY_ENDED.

130.5.6.2 public void failed(Coordination coordination) throws Exception

coordination The Coordination that has terminated as a failure.

□ Notification that a Coordination has terminated as a failure.

This Participant should discard any work associated with the specified Coordination.

Throws `Exception`– If this Participant throws an exception, the Coordinator service should log the exception. The Coordination.fail(Throwable) method which is notifying this Participant must continue notification of other registered Participants.

131 TR069 Connector Service Specification

Version 1.0

131.1 Introduction

This chapter provides a specification for the TR069 Connector, an assistant to a Protocol Adapter based on [1] *TR-069 Amendment 3*. A TR069 Connector provides a mapping of TR-069 concepts to/from the *Dmt Admin Service Specification* on page 365. It mainly handles the low level details of Object/Parameter Name to Dmt Admin URI mapping, and vice versa. TR-069 Protocol Adapter developers can use this service to simplify the use the Dmt Admin service. The TR069 Connector service is based on the definition of a Protocol Mapping in *Protocol Mapping* on page 408. It is assumed that the reader understands TR-069 and has a basic understanding of the Dmt Admin service.

The examples in this specification are not from a Broadband Forum Technical Report and are purely fictional.

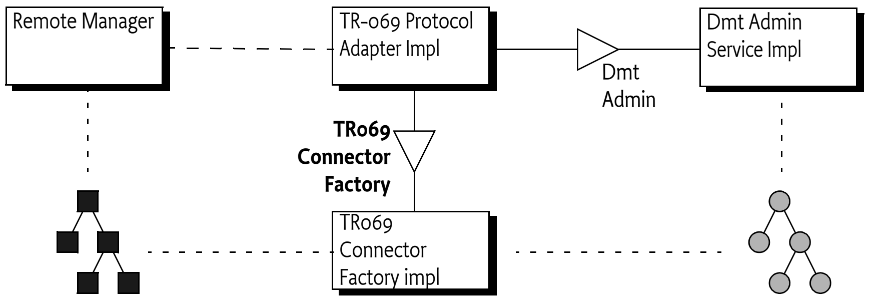
131.1.1 Essentials

- *Connector* - Provide a TR-069 view on top of the Dmt Admin service.
- *Simplify* - Simplify the handling of data models implemented through the DMT through the TR-069 protocol.
- *Browse* - Implement the constructs for MAP and LIST handling.
- *Native* - Provide a mechanism for Data Plugins to convey conversion information to the Protocol Adapter so that native TR-069 object models can be implemented as Data Plugins.

131.1.2 Entities

- TR069ConnectorFactory - Provides a way to create a TR069Connector that is bound to an active Dmt Session.
- TR069Connector - Created by TR069ConnectorFactory on a Dmt Session; provides methods that helps in using the TR-069 namespace and RPCs on a Dmt Admin DMT.
- ParameterValue - The value of a parameter, maps to the TR-069 ParameterValueStruct.
- ParameterInfo - Information about the parameter, maps to the TR-069 ParameterInfoStruct.
- DMT - The Device Management Tree as available through the Dmt Admin service.

Figure 131.1 TR-069 Entities



131.1.3 Synopsis

A TR-069 Protocol Adapter first creates a Dmt Session on the node in the DMT that maps to an object model that should be visible to the TR-069 Management Server. A Protocol Adapter can choose to map a whole sub-tree or it can create a virtual object model based on different nodes, this depends on the implementation of the Protocol Adapter.

When a TR-069 RPC arrives, the Protocol Adapter must parse the SOAP message and analyze the request. In general, an RPC can request the update or retrieval of multiple values. The Protocol Adapter must decompose these separate requests into single requests and execute them as a single unit. If the request is a retrieval or update of a data model maintained in the Dmt Admin service then the Protocol Adapter can use a TR069 Connector to simplify implementing this request. The TR069 Connector Factory service can be used to create an instance of a TR069 Connector that is based on a specific Dmt Session.

The TR069 Connector maps the Object or Parameter Name to a URI and perform the requested operation on the corresponding node. The name-to-URI conversion supports the LIST and MAP concepts as defined in *OSGi Object Modeling* on page 407.

The TR069 Connector handles conversion from the Dmt Admin data types to the TR-069 data types. There is a default mapping for the standard Dmt Admin formats including the comma separated list supported by TR-069. However, Data Plugins that implement TR-069 aware object models can instruct the TR069 Connector by providing specific MIME types on the Meta Node.

Objects can be added and deleted but are, in general, not added immediately. These objects are lazily created when they are accessed. The reason is that TR-069 does not support the concept of a session with atomic semantics, a fact leveraged by certain object models in the DMT. Therefore, adding an object will assign a instance id to an object but the creation of the object is delayed until the object is used.

After all the requests in an RPC are properly handled the TR069 Connector must be closed, the Dmt Session must be closed separately.

Errors are reported to the caller as they happen, if a Dmt Admin service error is fatal then the Dmt Session will be closed and it will be necessary to create a new TR069 Connector.

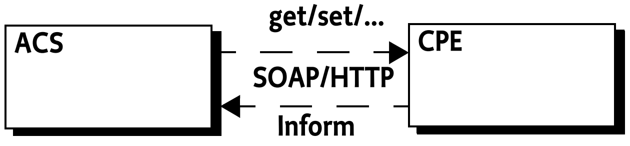
131.2 TR-069 Protocol Primer

The [6] *Broadband Forum* is an organization for broadband wire-line solutions. They develop multi-service broadband packet networking specifications addressing interoperability, architecture, and management. Their specifications enable home, business and converged broadband services, encompassing customer, access and backbone networks. One of the specifications of the Broadband Forum is the *Technical Report No 69*, also called TR-069, a specification of a management model.

131.2.1 Architecture

[1] *TR-069 Amendment 3* is a technical report (Broadband Forum's specification model) that specifies a management protocol based on [4] *SOAP 1.1* over HTTP. The TR-069 technical report defines a number of mandatory Remote Procedure Calls (RPCs) that allow a management system, the Auto-Configuration Server (ACS), to discover the capabilities of the Consumer Premises Equipment (CPE) and do basic management. This model is depicted in Figure 131.2.

Figure 131.2 *TR-069 Reference Architecture*



In TR-069, the CPE is always initiating the conversation with the ACS though the ACS can request a session.

Inside the CPE there is a Protocol Adapter that implements the TR-069 RPCs. These RPCs read and modify the objects models present in the CPE. There is usually a mechanism that allows the different modules in the CPE to contribute a management object to the Protocol Adapter so that the Protocol Adapter does not require knowledge about highly specialized domains.

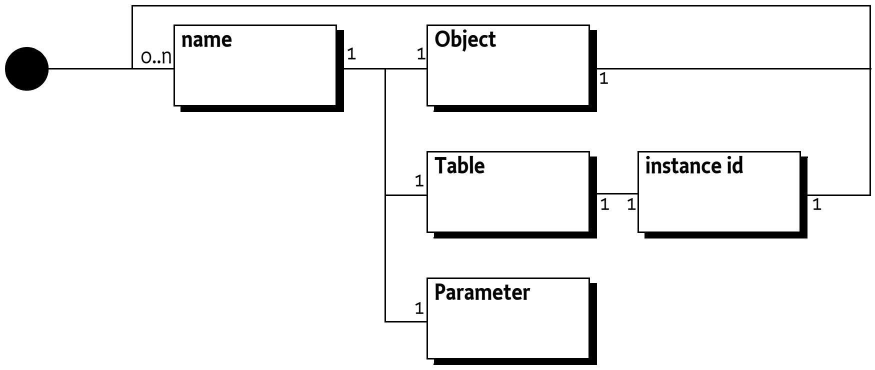
[2] *TR-106 Amendment 3* specifies object model guidelines to be followed by all TR-069-enabled devices as well as a formal model to document these object models.

131.2.2 Object Model

The object model of TR-069 consists of *objects* that contain *parameters* as well as *tables* that contain objects. TR-106 says:

- *Object* - A named collection of parameters and/or other objects.
- *Parameter* - A name-value pair.
- *Table* - An enumeration of objects identified by an instance id.

Figure 131.3 *Type Model TR-069*



Objects can also occur in tables, in that case the object name is suffixed with an *instance id*. An object that has no instance id is a singleton, with an instance id they are referred to as *tables*. In the Broadband Forum technical reports tables end in the special suffix {i}, the instance id.

This provides the following structural definition for this specification:

```
named-value ::= NAME ( object | table | parameter )
```

```
object      ::= named-value +
table       ::= ( instance object )*
parameter   ::=
instance    ::= INTEGER > 0
```

TR-069 talks about partial paths and parameter names. In this specification, a *name* is reserved for the short relative name used inside an object, also called the *local name*. The term *path* is reserved for the combination of object names, table names, and instance ids that are separated by a full stop ('.' \u002E) and used to traverse an instance model.

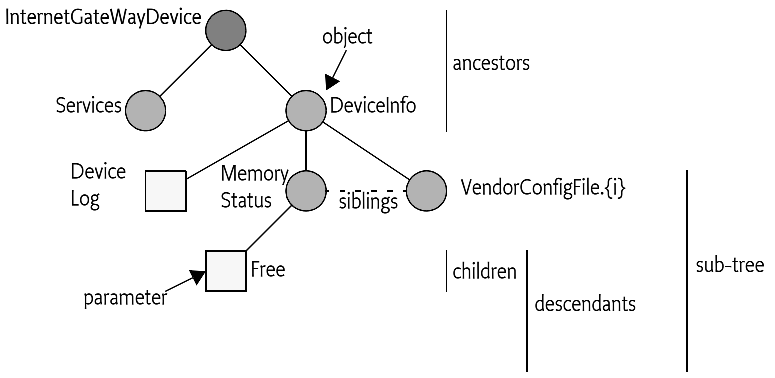
```
path           ::= parameter-path | object-path| table-path
segment        ::= NAME '.' ( instance '.' )?
object-path    ::= segment+
table-path     ::= segment* NAME '.'   // expect INTEGER next
parameter-path ::= object-path NAME
instance-path  ::= table-path instance '.'
```

In this specification the following terms are used consistently:

- *Object* - Refers to a named type defining a certain set of parameters, objects, and tables.
- *Table* - A list of instances for a given object.
- *Instance* - An object element in a table at a certain id.
- *Instance Id* - The integer id used to identify an instance in a table.
- *Alias* - A name chosen by the ACS that uniquely identifies an instance.
- *Singleton* - An object that is not in a table.
- *Name* - The name of an object, table, or parameter refers to the local name only and not the path.
- *Segment* - A component in a path that always ends in a full stop. A segment can contain instance ids to identify an instance.
- *Path* - A string uniquely identifying a path in the tree to either a parameter, an object, or a table.
- *Object Path* - A path that uniquely identifies an instance or a singleton. An object path must always ends in a full stop. This maps to the TR-069 concept of an `ObjectName`.
- *Parameter Path* - The name of the parameter preceded by the owning object. A path that does not end in a full stop is always a parameter path.
- *Table Path* - An object path that lacks the last instance id. In TR-069 this is also sometimes called a partial path. The last segment is an object path that must be followed by an instance id to address an instance.
- *Instance Path* - A path to an instance in a table

This provides a hierarchy as depicted in Figure 131.4.

Figure 131.4 *TR-069 Object and Parameter naming relative to the parameter MemoryStatus*



131.2.3 Parameter Names

The grammars for parameter names and object names are as follows:

```
NAME ::= ( Letter | '_' )
         ( Letter | Digit | '-' | '_' | CombiningChar| Extender )*
```

The productions Letter, Digit, CombiningChar, and Extender are defined in [5] *Extensible Markup Language (XML) 1.0 (Second Edition)*. The name basically supports the full Unicode character set for letters and digits (including digits for other languages), including sets for languages like Hebrew and Chinese. Examples of different parameter names are:

```
name            // simple name
Name            // case sensitive

_
_-_-_
ångstrom
þingsten
ΨΣΩΠ
```

131.2.4 Parameter Type

A parameter value can have one of the data types defined in [2] *TR-106 Amendment 3*, they are summarized in Table 131.1.

Table 131.1 TR-106 Data types

TR-106 Type	Description
object	Represents a structured type
string	A Unicode string, optionally restricted in length
int	32 bit integer
long	64 bit integer
unsignedInt	32 bit unsigned integer
unsignedLong	64 bit unsigned integer
boolean	Can have values 0 or false (false) or 1 or true (true)
dateTime	TR-069 recognizes three different date times. These three cases are differentiated in the following way: • *Unknown time* - If the time is not known. • *Relative time* - Relative time is the time since boot time. • *Absolute time* - Normal date and time.
base64	An array of bytes
hexBinary	An array of bytes

SOAP messages always provide a type for the parameter value. For example:

```
<ParameterValueStruct>
  <name>Parameter1</name>
  <value xsi:type="long">1234</value>
</ParameterValueStruct>
```

The xsi prefix refers to the https://www.w3.org/2001/XMLSchema-instance namespace. However, this makes not all TR-106 types well defined, for example in XML Schema base64 is called base64Binary. This specification assumes that the names and definitions in Table 131.1 and provides appropriate constants for the Protocol Adapter.

Parameters can be read-only or read-write. All writable Parameters must also be readable although security can cause certain parameters to be read as an empty string, for example passwords. Parameters can reflect configuration as well as status of the device. External causes can cause parameters to change at any time. The TR-069 protocol has the facility to call an Inform RPC to provide the ACS with a notification of changed parameters.

131.2.5 Parameter Attributes

Parameter attributes provide the meta data for a parameter. In TR-069, the attributes are used to manage notifications and access control. Each parameter in TR-069 can be watched by the ACS by setting the corresponding parameter attribute to *active* or *passive notifications*. Passive notifications are passed whenever the CPE communicates with the ACS and active notifications initiate a session. Parameters that have a notification are said to be *watched*.

Access to the parameters can be managed by setting Access Control Lists via the corresponding parameter attribute.

131.2.6 Objects and Tables

TR-106 has the concept of an *object* stored in a *table* to allow multiple instances of the same type. It is part of the object definition if it is stored in a table or not. An object cannot both appear as a table instance and as a singleton.

Each instance in the table is addressed with an integer >= 1. This *instance id* is not chosen by the ACS since it can be required to create a new instance due to an external event. For example the user plugging in a USB device or starting a new VOIP session. The ACS must discover these instance ids by asking the device for the instance ids in a table.

For example, the parameter path Device.LAN.DHCPOption.4.Request refers to a parameter on a DHCPOption object that has the instance id 4. Instance ids are not sequential nor predictable. It is the responsibility of the device to choose an instance id when an object is created. Instance ids are assumed to be persistent so that the ACS can cache results from a discovery process.

Newer TR-069 objects have been given an Alias parameter. This alias uniquely identifies the table instance.

TR-069 defines a convention for a parameter that contains the number of entries in a table. Any parameter name that ends with NumberOfEntries contains the number of entries in a table with the name of the prefix in the same object. For example A.B.CNumberOfEntries provides the number of entries in the table:

```
A.B.C.
```

131.2.7 RPCs

The object model implemented in a device is accessed and modified with *RPCs*. RPCs are remote procedure calls; a way to invoke a function remotely. TR-069 defines a number of mandatory RPCs and provides a mechanism to extend and discover the set of RPCs implemented by a CPE. The mandatory RPCs are listed in in the following table.

Table 131.2 *TR-069 RPCs*

RPC	Description
GetRPCMethods	Return a list of RPC methods
SetParameterValues	Set one or more parameter values
GetParameterValues	Get one or more parameter values

RPC	Description
GetParameterNames	Get the parameter information for a parameter, object, or table.
SetParameterAttributes	Set parameter attributes
GetParameterAttributes	Get parameter attributes
AddObject	Add a new object to a table
DeleteObject	Delete an object from a table
Download	Download software/firmware
Reboot	Reboot the device

131.2.8 Authentication

The security model of TR-069 is based around the authentication taking place during the setup of a TLS (formerly SSL) connection. This authentication is then used to manage the access control lists via the parameter attributes.

131.2.9 Sessions and Transactions

A *session* with the ACS is always initiated by the CPE. The ACS can request a session, but it is always the CPE that starts a session by opening the connection to the ACS and then sending an Inform RPC. The session ends when the connection is closed, which happens after the ACS has informed the CPE it has no more requests.

During a session, a CPE has the requirements that parameters must not change due to other sources than the session and that the parameters are consistent with the changes. However, there is no transactionality over the session, atomicity is only guaranteed for one RPC. An RPC can consist of multiple parameter modifications that should therefore be atomically applied.

131.2.10 Events and Notifications

TR-069 sessions always start with an Inform RPC from the CPE to the ACS. This RPC contains any events and notifications for parameters that were watched. Events signal crucial state changes from the CPE to the ACS. For example, if a device has rebooted it will inform the ACS. Notifications are caused by parameter changes, the Inform RPC contains a list of events and parameters with changed values.

131.2.11 Errors

Invoked RPCs can return a fault status if errors occur during the execution of the RPC. For ACS to CPE RPCs these fault codes start at 9000, for the reverse direction they start at 8000. Each RPC defines the fault codes that can occur and their semantics in that context.

131.3 TR069 Connector

A TR-069 Protocol Adapter must be able to browse foreign Data Plugins on the device and support native TR069 objects models implemented by a Data Plugin. As Data Plugins are available through the Dmt Admin service, the Protocol Adapter must provide a bi-directional mapping between Dmt Admin nodes and TR-069 parameters, notifications, and error codes.The mapping must enable a Data Plugin to provide a native Broadband Forum object model that limits itself to the required RPCs.

131.3.1 Role

Developers implementing the TR-069 protocol are not likely to be also experts in the Dmt Admin service. This specification therefore provides a TR069 Connector Factory service that provides an object that can map from the TR-069 concepts to the Dmt Admin concepts, supporting all the constructs defined in the *OSGi Object Modeling* on page 407.

The TR069 Connector only specifies a number of primitive functions to manage the DMT. Parsing the SOAP messages, handling the notifications, and splitting the requests for TR069 Connector is the responsibility of the Protocol Adapter. The reason that the TR069 Connector does not work on a higher level is that a Protocol Adapter for TR-069 will likely communicate with other subsystems in the CPE than the OSGi framework alone. Though the Dmt Plugin model is an attractive approach to implement object models, there is history. Existing code will likely not be rewritten just because it can be done better as a Data Plugin.

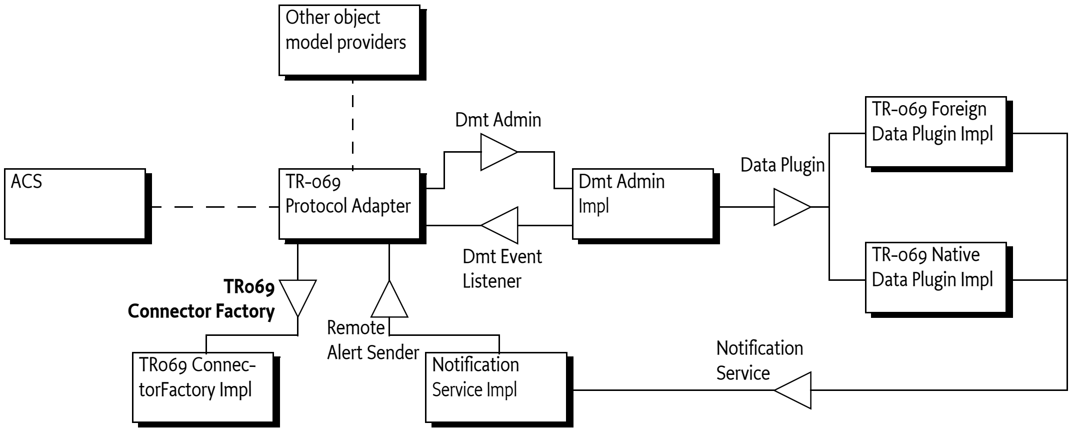
For example, a Data Plugin could implement the `Device.DeviceInfo.` object. However, this object actually resides in the DMT at a node:

```
./TR-069/Device/DeviceInfo
```

A TR-069 Protocol Adapter will therefore be confronted with a number of data models that reside in different places. Each place provides one or more consistent data models but it is the responsibility of the TR-069 Protocol Adapter to ensure the ACS gets a consistent and standardized view of the whole. To create this consistent view it will be necessary to adapt the paths given in the RPCs. It is expected that a Protocol Adapter is required to have a certain amount of domain knowledge, for example a table, that maps TR-069 paths to their actual providers.

The basic model is depicted in Figure 131.5.

Figure 131.5 TR-069 Connector Context


Other object model providers
Dmt Admin
TR-069 Foreign Data Plugin Impl
Data Plugin
ACS
TR-069 Protocol Adapter
Dmt Admin Impl
Dmt Event Listener
TR-069 Native Data Plugin Impl
TR069 Connector Factory
Remote Alert Sender
Notification Service
TR069 ConnectorFactory Impl
Notification Service Impl

The Protocol Adapter can be implemented as an OSGi Bundle or it can be implemented in native code in the device. Both architectures are viable. For certain aspects like the TR-157a3 Software Modules a certain amount of native code will be required to manage the OSGi Framework as an Execution Environment.

In an environment where the Protocol Adapter is implemented outside an OSGi Framework it will be necessary to create a link to the Dmt Admin service. This can be achieved with a proxy bundle inside the OSGi framework that dispatches any requests from the native Protocol Adapter to the functionality present in the OSGi Framework. In this specification, it is assumed that such proxies can be present. However, the examples are all assuming that the Protocol Adapter is running as a Bundle.

131.3.2 Obtaining a TR069 Connector

A TR069 Connector is associated with a Dmt Session, the `TR069ConnectorFactory` provides the `create(DmtSession)` method that will return a `TR069Connector` object. This object remains associated with the Dmt Session until the Dmt Session is closed, which can happen because of a fatal error or when the TR069 Connector Factory is unregistered or un-gotten/released. Creating a TR069 Connector must not be expensive, Protocol Adapters should create and close them at will. Closing the connector must not close the corresponding Dmt Session.

The TR069 Connector must use the root of the session as its *base*. That is, their URI mapping all parameters must start from the base. For example, if the session is opened at ./TR-069 then the parameter IGD/DeviceInfo/Manufacturer must map to URI ./TR-069/IGD/DeviceInfo/Manufacturer.

If a Protocol Adapter will modify the tree then it should use an atomic session for all RPCs even if the RPC indicates read-only. The reason for the atomicity is that in certain cases the lazy behavior of the TR069 Connector requires the creation of objects during a read operation. If a non-atomic session is used then the TR069 Connector must not attempt to lazily create objects and reject any addObject(String) and deleteObject(String) methods. See also *Lazy and Sessions* on page 653.

131.3.3 Supported RPCs

The TR069 Connector supports a limited number of RPCs, and for those RPCs it only supports the singleton case. The TR069 Connector provides support for the RPCs primitives listed in the following table.

Table 131.3 Supported TR-069 RPCs

RPC	Related Method	Description
SetParameterValues	setParameterValue(String,String,int)	Set one or more parameter values. The connector supports setting a single value, ensuring the proper path traversal and data type conversion
GetParameterValues	getParameterValue(String)	Get one or more parameter values. The connector supports getting a single value, converting it to a ParameterValue object, which contains the value and the type.
GetParameterNames	getParameterNames(String,boolean)	Get the paths of objects and parameters from the subtree or children that begins at the parameter path. The TR-069 Connector supports the full traversal of the given path and the next level option.
AddObject	addObject(String)	Add a new object to a table. The fully supports the semantics, taking the MAP and LIST nodes into account. Node creation can be delayed until a node is really needed.
DeleteObject	deleteObject(String)	Delete an object from a table.

131.3.4 Name Escaping

An object or parameter path describes a traversal through a set of objects, this is almost the same model that Dmt Admin provides. The difference is that the characters allowed in a TR-069 parameter name are different from the Dmt Admin node names and that TR-069 does not support application specific parameter/object names like the Dmt Admin service does.

A path consist of a number segments, where each segment identifies a name or instance id. TR-069 names can always be mapped to Dmt Admin node names as the character set of TR-069 parameter names is restricted and falls within the character set of the Dmt Admin node names. The length of a segment could be a problem but TR-069 paths are generally limited to have a length of less than 256 bytes. This specification therefore assumes that a segment of a TR-069 path is never too long to fit in a Dmt Admin node name.

Mapping a Dmt Admin node name to a parameter name, needed for browsing, is more complicated as Dmt Admin node names allow virtually every Unicode character except the solidus ('/' \u002F). It is therefore necessary to escape Dmt Admin URIs into a path that is acceptable for the TR-069 protocol. It is assumed that escaping is only used in a browsing mode since native object models will never require escaping. The TR069 Connector must return names from the `getParameterNames(String,boolean)` call that the ACS can handle, optionally show to the user, and then use to construct new paths for subsequent RPCs.

There is no obvious escape character defined in TR-069, like for example the reverse solidus ('\' \u005C) that the Dmt Admin uses for escaping. The character for escaping is the latin small letter thorn ('þ' \u00FE) because his character is highly unlikely to ever be used in a TR-069 path for a native object model, however, even if it is then it would be no problem for the escaping algorithm. The thorn is a letter, allowing it to be used as the first character in a parameter name, this allows escaping the first character.

A character in a segment that is not allowed must be escaped into the following sequence:

```
þ[0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z]
```

The 4 hexadecimal upper case digits form a hexadecimal number that is the Unicode for that character. Each character that does not conform to the syntax specified in *Parameter Names* on page 645 or the thorn character itself must be replaced with the escape sequence. For example, the name 3ABCþ must be translated to:

```
þ0033ABCþ00FE
```

If the segment is an instance id then the segment must not be escaped. Otherwise, if the segment does not start with a `Letter` or underscore, then the first character must be escaped with the thorn.

Unescaping must undo the escaping. Any sequence of þ[0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z] must be replaced with the character with the corresponding Unicode. A thorn found without the subsequent 4 hexadecimal upper case digits must be treated as a single thorn. For readability it is best to minimize the escaping. However, any name given to the TR069 Connector that is escaped must be properly interpreted even if the unescaped string did not require escaping. For example, þ0031þ0032þ0033 must be usable as an object instance id as the unescaped form is 123, which is a number.

A number of examples of the escaping are shown in the following table.

Table 131.4 *Escaping Parameter Names*

Segment	Dmt Admin Escaped	TR-069 Escaped	Notes
DeviceInfo	DeviceInfo	DeviceInfo	Most common case.
3x Hello World	3x Hello World	þ0033xþ0020Helloþ0020World	The initial digit and the spaces must be escaped in TR-069.
þorn	þorn	þornþ00FEorn	A single thorn does not require escaping as it is not followed by 4 hexadecimal digits. So both forms are valid for unescaping although escaping must deliver the þ00FE form.
application/bin	application\/bin	applicationþ002Fbin	The solidus must be escaped in both.
234	234	234	A numeral does not require escaping, it is assumed to be an instance id.
234x	234x	þ003234x	A name that starts with a digit requires the first digit to be escaped.
þ00FEorn	þ00FEorn	þ00FE00FEorn	It is possible to encode even already escaped names.

The TR069 Connector only accepts escaped paths and returns escaped paths. When a method returns a path it must be properly escaped and suitable as a TR-069 path.

131.3.5 Root

In general, the TR-069 Protocol Adapter is free to choose what parts of the DMT it wants to expose. A simple mapping table containing path prefixes can be used to define the handler for the given data model. However, since the intention is to allow TR-069 object models to be implemented in Dmt Admin Data Plugins there is a need to know where those plugins should reside in the DMT. This root is defined as:

```
./TR-069
```

Any Data Plugin that wants to provide an object model in the TR-069 family of object models should provide a Data Plugin rooted at the TR-069 root. For example, a Data Plugin implementing the InternetGatewayDevice.DeviceInfo. object should register its Data Plugin under the data Root URI ./TR-069/ InternetGatewayDevice/DeviceInfo

131.3.6 DMT Traversal

A path must be mapped from the TR-069 hierarchy to the Dmt Admin nodes URI. The Protocol Adapter decides the *base* in the DMT by opening the Dmt Session with a session root parameter. The TR-069 Connector must then traverse the tree from this base based on the TR-069 path. The Protocol Adapter must use the *Instance Id* on page 414 for MAP and LIST nodes to traverse the DMT.

Assume that the URI of a node is requested for a given path P. The path P must be traversed from the root node. The root node can find the child, the first segment in P, and then use the same routine recursively for the remainder. This recursive routine must perform the following actions on each current node:

- If path P is empty, then this is the requested node.
- S = first segment of path P up to the first full stop.
- R = remainder of path P after the first full stop or empty if no full stop.
- If S is an alias (surrounded by '[' and ']'), replace S with the alias inside the brackets. For Dmt Admin nodes aliases are identical to normal node names.
- unescape S (replace the thorns)
- If the current node is a MAP or a LIST and S is an integer
  - Get the list L of children of the current nodes
  - If the nodes in L have an InstanceId node find the node where the InstanceId matches the segment S as integer, this becomes then the next level node N and the algorithm is repeated with path R.
- If no next node N was found then make it the child node of the current node with the name S.
- Repeat the algorithm with N with path R

Since each node that is traversed this way knows the node name it corresponds to it is easy to create an encoded URI for Dmt Admin.

For example, the TR-069 path:

```
Device.DeviceInfo.Interface.14.Connections.3.BytesSent
```

Assuming that Interface node is a MAP node and its children have an InstanceId node, where the WAN_1 node has an InstanceId of 14.

The Connections node is a LIST and the children have no InstanceId, therefore the name is the index. The translated URI then looks like:

```
Device/DeviceInfo/Interface/WAN_1/Connections/3/BytesSent
```

The toURI(String,boolean) method can take a TR-069 path and perform the substitutions. If the create parameter is true then the TR069 Connector will create *missing nodes* if possible. Missing nodes can only be created under a LIST or MAP node.

A missing node is a node that is addressed by a path but not present in the DMT. For example, the root of the session is ./TR-069 and the parameter path is A.B.C. If the DMT contains ./TR-069/A but not ./TR-069/A/B then node B is a missing node.

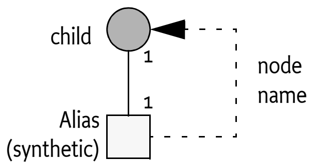
131.3.7 Synthetic Nodes

The Protocol Adapter must synthesize an Alias parameter and for any MAP or LIST node called X it must provide a sibling XNumberOfEntries parameter that provides the number of entries in table X.

131.3.7.1 Alias

The Alias node is a read-write parameter that must map to the actual node name of its parent. For example, ./A/B/C/Alias must map to C. Reading it must provide the this parent's node name and writing it must rename this parent's node name. The Alias must be automatically provided on any child of a MAP node. The Alias parameter must also be returned in the result of getParameterNames(String,boolean) if its parent's children are included. It is not possible to convert an Alias parameter name to a URI as the Alias node is synthetic and does not exist in the DMT. The model of aliases are depicted in Figure 131.6.

Figure 131.6 Aliases



Aliases can be used by the ACS to set the key of a MAP. For example, if a set of properties is defined as a MAP:

Name	Act	Type	Card.	S	Description
Properties	Get	MAP	1	P	A Properties map
[string]	Get Set Add Del	string	0..n	A	Key/Value

An ACS can first add an object to the table. This will create an entry with a calculated instance id. However, the ACS can then rename the node with the Alias node. In pseudo code:

```
AddObject          ..Properties.                (returnsnode name = 3421)
SetParameterValue  ..Properties.3421.Alias = MyKey
```

Alternatively, addressing with an alias in the parameter name would be simpler:

```
AddObject          ..Properties.[MyKey]
```

131.3.7.2 Number Of Entries

TR-069 has the convention of parameters that end with NumberOfEntries. For example, the parameter UserNumberOfEntries in the object InternetGatewayDevice object contains the number of entries of the InternetGatewayDevice.User table.

The Protocol Adapter must synthesize these NumberOfEntries parameters for each MAP or LIST node. The NumberOfEntries parameter must be a sibling of the MAP or LIST node. Any such parameter must also be returned in the result of the getParameterNames(String,boolean) method.

131.3.8 Lazy and Sessions

In the Dmt Admin service the session plays an important role in how the object model operates. Especially atomic sessions have a clear point to commit any changes so that many actions can be deferred until all the information is available. In TR-069 there is no real session concept although one RPC must be executed atomically even if it changes multiple parameters. As there are different RPCs to create objects and set their parameters it is impossible to create and parameterize an object in a single session. This creates problems with general DMT models.

It is recommended to operate all RPCs in an atomic session to allow these DMT models to leverage the session commit phase. However, a TR-069 Connector must also accept a read only or exclusive session. The session can then of course cause exceptions to be thrown at certain operations.

The connector must *lazily* create instances. An addObject(String) method must not actually create the object, it only has to create an instance id and ensure the uniqueness of this id over time. The id must follow the rules from TR-069, it must not clash with an existing id even after such an id has been used in the past.

This id is then returned to the ACS who will then use it in subsequent RPCs. When one of the subsequent RPCs tries to access this not-yet existent node, for example a get or set, then the TR069 Connector must create it before it sets or gets the value of this node. This lazy strategy allows the node creation and the parameterization of that node to happen in a single session/RPC.

For example, in session 100 the addObject(String) creates a new node. This node is not really created but the unique instance id 4311 is assigned to it. After this RPC, the session is closed. The ACS receives this instance and then prepares a GetParameterValues RPC to get the ../4311/Foo parameter. The management agent receives the RPC and opens a new session 200, it then calls getParameterValue(String). The TR069 Connector will not find the appropriate entry 4311 in the table. Instead of raising an error it creates this node and then gets the value for the ../4311/Foo parameter.

A Data Plugin implementing a native TR-069 object model can override the lazy behavior by adding a application/x-tr-069-eager MIME type to the list of MIME types in the Meta Node. If this MIME type is present then the node must be eagerly created during the addObject(String) method.

The TR069 Connector must assign the unique id according to the TR-069 rules for instance ids.

131.3.9 Data Types

This specifications assume the [2] *TR-106 Amendment 3* defined data types. TR-106 defines a number of data types, derived from XML Schema and creates a number of sub-types to discriminate between different use cases. A Protocol Adapter must be able to understand the types defined in Table 131.5 to be able to faithfully define a data model based on [2] *TR-106 Amendment 3*. Discriminating between some of the sub-types requires inspection of the data. Each sub-type requires mapping rules that are defined later. Each mapping is assigned a unique MIME sub-type in the application media type. That is, the TR-069 int type has a MIME type of application/x-tr-069-int.

Table 131.5 TR-069 Types, MIME types

TR-069 Type	MIME Type	Notes
base64	x-tr-069-base64	Base 64 encoded
hexBinary	x-tr-069-hexBinary	Hex encoded
boolean	x-tr-069-boolean	
string	x-tr-069-string	General string type.
string (list)	x-tr-069-list	A comma separated string that acts as a list.
int	x-tr-069-int	Signed integer
unsignedInt	x-tr-069-unsignedInt	Unsigned integer
long	x-tr-069-long	Signed long

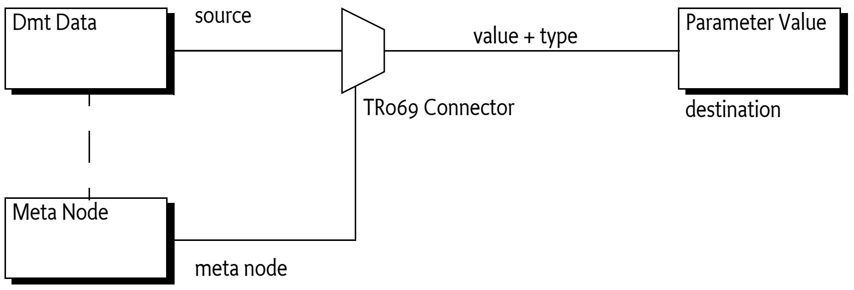
TR-069 Type	MIME Type	Notes
`unsignedLong`	`x-tr-069-unsignedLong`	Unsigned long
`dateTime`	`x-tr-069-dateTime`	Absolute UTC time, relative boot time, or unknown time
	`x-tr-069-eager`	Eager creation (not a data type, see *Lazy and Sessions* on page 653).

It is the responsibility of the Protocol Adapter to properly clean up the parameter values, that is, remove any unnecessary white space, etc. The TR069 Connector must accept any lexically correct form of the value of a parameter. However, the connector must always return the value according to the format of the data types specified by TR-069.

131.3.10 DMT to TR-069 Conversion

This section describes the conversion from a DMT node (a Dmt Data) to a TR-069 Parameter value. The *source* is the DMT node retrieved from the DMT. The *destination* is the value and its type that must be encoded in the TR-069 response. The *meta node* is the Meta Node associated with the source. This model is depicted in Figure 131.7.

Figure 131.7 DMT to TR-069



The different conversions possible for the Dmt Data to the TR-069 Parameter value are shown in Table 131.6. This table shows vertically the Dmt Admin formats and horizontally the TR-106 types defined in Table 131.5. Each row has a default conversion type, indicated with a bold entry. For example, the default conversion of a `FORMAT_BOOLEAN` to the `boolean` type is the default conversion.

This default conversion can be overridden by the Data Plugin by specifying an alternative MIME type in the list of allowed MIME types in the Meta Node `getMimeTypes()`. If this list contains a MIME type that has the prefix `application/x-tr-069-` then the first entry in this list must be chosen as the destination type instead of the default type. This way, a TR-069 Data Plugin can indicate the exact type to a TR-069 Protocol Adapter.

For example, a Dmt Data has the format `FORMAT_BASE64`. However, the Data Plugin for this node has a Meta Node that contains

```
String[] { "application/x-tr-069-hexBinary"}
```

The resulting type must therefore be `hexBinary` in this example.

The Dmt Data nodes are leaf nodes, however, there is a special case for interior `LIST` nodes marked with a `application/x-tr-069-list` type in the Meta Node. These nodes must be converted to a comma separated string as described in *List* on page 656.

Cells that are empty in the table indicate an impossible conversion that must be reported. Cells with a name refer to one of the subsequent sections.

Table 131.6 Dmt Data Format to TR-069 Data

	base64	**boolean**	**dateTime**	**hexBinary**	**int**	**long**	**string**	**unsignedInt**	**unsignedLong**
FORMAT_BASE64	binary			binary					
FORMAT_BINARY	**binary**			binary					
FORMAT_BOOLEAN		=					true | false		
FORMAT_DATE			**date**				=		
FORMAT_DATE_TIME			**date**				date		
FORMAT_FLOAT					number	**number**	number	number	number
FORMAT_INTEGER					**number**	number	number	number	number
FORMAT_LONG					number	**number**	number	number	number
LIST							**list**		
FORMAT_NULL		false	date		0	0	"null"	0	0
FORMAT_RAW_BINARY	**binary**			binary					
FORMAT_RAW_STRING							=		
FORMAT_STRING							=		
FORMAT_TIME			date				=		
FORMAT_XML							=		

131.3.10.1 Date

If the destination type is string then a date must be formatted according to the TR-069 dateTime format. FORMAT_DATE and FORMAT_TIME must be set to a TR069_DATETIME typed destination with just the day or just the time respectively. That is, the FORMAT_TIME must be treated as a relative time for TR-069.

The Date object of the Dmt Data object represents the three different TR069_DATETIME types with the getTime() method. The value of getTime() indicates what type of date time it is:

- *Unknown* - The getTime() method must be 0
- *Relative* - The getTime() method must return a negative number
- *Absolute* - The getTime() method must return a positive number

If a FORMAT_DATE, FORMAT_TIME, or FORMAT_DATE_TIME is converted to a string the string representation of TR069_DATETIME must be used, including the form of unknown, relative, or absolute. A FORMAT_NULL stands for an unknown time.

131.3.10.2 Binary

The Dmt Admin service has several binary formats (FORMAT_BASE64, FORMAT_BINARY, and FORMAT_RAW_BINARY) that can be converted to TR069_HEXBINARY and TR069_BASE64. All binary formats maintain their data as a byte[]. Conversion is therefore straightforward encoding of the byte[] into the proper encoding: hex or base 64.

131.3.10.3 Number

The TR-069 Connector must convert numeric values (FORMAT_INTEGER, FORMAT_LONG, and FORMAT_FLOAT) to TR069_INT, TR069_LONG, TR069_UNSIGNED_INT, and TR069_UNSIGNED_LONG values. Float values must be rounded according to the standard Java rounding rules when converted to an integer or long.

A conversion must not exceed the range of the destination type. That is, if an integer is converted to an unsigned int then negative values must be treated as an error. If the destination type is string then the numeric value must be calculated with the Dmt Data toString method.

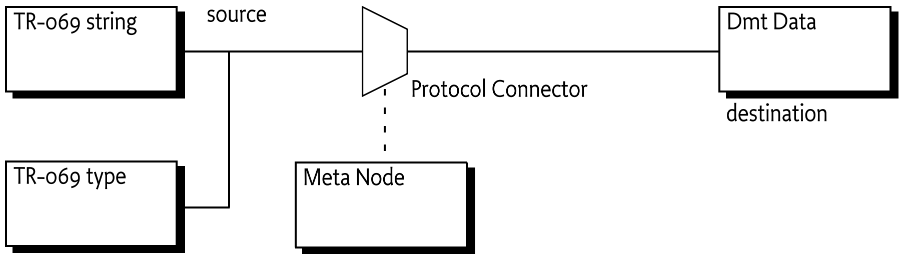
131.3.10.4 List

LIST nodes with primitive children must be converted to a comma separated list. If the children nodes are interior nodes then an error must be raised. The values of the comma separated list must come from the children of the value node. Each of these children must be converted to a string type according to Table 131.6. These children must then be escaped and concatenated with a comma as separator according to the rules of TR-106 comma separated lists. Nested lists are not allowed.

131.3.11 TR-069 to Dmt Data Conversion

A TR-069 Parameter value consists of a string and a type identifier from the set of TR-069 types, see *Data Types* on page 653. The conversion is depicted in Figure 131.7.

Figure 131.8 *TR-069 to DMT*



The destination type is obtained from the corresponding Meta Node. If multiple formats are specified in the result of the getFormat() method then the most applicable type must be used. The following table lists the applicability for each TR-106 data type.

base64	FORMAT_BASE64, FORMAT_BINARY, FORMAT_RAW_BINARY
boolean	FORMAT_BOOLEAN, FORMAT_STRING
dateTime	FORMAT_DATE_TIME, FORMAT_DATE, FORMAT_TIME
hexBinary	FORMAT_BASE64, FORMAT_BINARY, FORMAT_RAW_BINARY
int	FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_STRING
long	FORMAT_LONG, FORMAT_FLOAT, FORMAT_INTEGER, FORMAT_STRING
string	FORMAT_STRING, FORMAT_BOOLEAN, FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_RAW_STRING, FORMAT_XML
unsignedInt	FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_STRING
unsignedLong	FORMAT_LONG, FORMAT_FLOAT, FORMAT_INTEGER, FORMAT_STRING

If the conversion fails and there are untried formats left then the other formats must be used.

There is a special case when the destination node is a LIST node with primitive children and the source is a string type. In that case the string must be parsed according to TR-106 comma separated lists and each element must be stored as a child node.

The conversion matrix is in the following table. The equal sign indicates identity taking into account any encoding. It is not necessary that the source type corresponds to a MIME type in the meta node.

Table 131.7 TR-069 Value to Dmt Data

	FORMAT_BASE64	FORMAT_BINARY	FORMAT_BOOLEAN	FORMAT_DATE	FORMAT_DATE_TIME	FORMAT_FLOAT	FORMAT_INTEGER	FORMAT_LONG	FORMAT_RAW_BINARY	FORMAT_RAW_STRING	FORMAT_STRING	FORMAT_TIME	FORMAT_XML	LIST
base64	binary	binary							binary					
boolean			bool								true| false			
dateTime				date	date						=	date		
hexBinary	binary	binary							binary					
int						num	num	num			=			
long						num	num	num			=			
string			bool			num	num	num		=	=		=	list
unsignedInt						num	num	num			=			
unsignedLong						num	num	num			=			

131.3.11.1 Date

A TR069_DATETIME can be converted to a FORMAT_DATE, FORMAT_TIME, and FORMAT_DATE_TIME. A FORMAT_DATE must take the day part and a FORMAT_TIME must take the time part.

131.3.11.2 Num

Source numbers must be converted to their destination counterpart. The conversion result must fail if the result falls outside the range of the destination.

131.3.11.3 Bool

If the source is a string or boolean type and the destination FORMAT_BOOLEAN then the conversion must parse the string ignoring the case. The strings true and false map to their corresponding value. The strings 0 must map to false and 1 to true.

131.3.11.4 Binary

The source must be decoded according to its TR-069 type (TR069_BASE64 or TR069_HEXBINARY). The resulting byte array can then be set with the DmtData(byte[],int) with the destination format: FORMAT_BINARY or FORMAT_BASE64.

131.3.11.5 List

The source is a comma separated list and must be stored as children of the destination node.

131.4 RPCs

The following sections explain in more detail how the different RPCs are supported by the TR069 Connector operate.

131.4.1 Get Parameter Values

The GetParameterValues RPC retrieves the value from one or more parameters. Each request in the RPC can request one parameter value or provides an object or table path, requesting multiple values with one path.

The getParameterValue(String) method retrieves the value of one parameter in the DMT. The getParameterNames(String,boolean) method can be used to retrieve the values of a table or object.

For the getParameterValue(String) method the TR069 Connector must first check for synthesized parameters, see *Synthetic Nodes* on page 652 (Alias and NumberOfEntries). Otherwise, the parameter name must be converted to a URI, this must be done according to the toURI(String,boolean) method with the boolean set to true, creating any missing nodes if possible. The Dmt Data for this node must be converted according to *DMT to TR-069 Conversion* on page 654. The returned ParameterValue contains the type and value of the parameter.

For example:

```
ParameterValue v = connector.getParameterValue(
          "Device.DeviceInfo.Manufacturer");
String value = v.getValue();
int type = v.getType();
```

131.4.2 Set Parameter Values

The SetParameterValues RPC sets a number of values in one RPC. The setParameterValue(String,String,int) method corresponds to setting a single parameter in the DMT. It takes a parameter path, a value, and the type of this parameter.

The TR069 Connector must first check if the requested destination is the Alias node of a MAP child. If the Alias node is set, the name of the parent node must be renamed to the given value. The value of the Alias node must be a TR-069 string type, the Connector must ensure the value is escaped when necessary. See *Synthetic Nodes* on page 652 for further information about aliases.

Otherwise, the parameter name must be converted to a URI, this must be done according to the toURI(String,boolean) method with the boolean set to true.

The given value must be converted to a Dmt Data according to the *TR-069 to Dmt Data Conversion* on page 656. For example:

```
connector.setParameterValue("Starwars.R2D.2.Start",
                             "20110805T10:15:20Z", TR069_DATETIME );
```

131.4.3 Get Parameter Names

The GetParameterNames RPC allows an ACS to discover the parameters accessible on a particular CPE as well as verifying the existence of a parameter. There are modes for this RPC depending on the path and next level arguments. See the following table.

Table 131.8 Modes based on type of path and NextLevel arguments

NextLevel	Parameter Path	Table or Object Path
true	Invalid Argument Fault code 9003 since this field must always be false for a parameter path.	Include only the children of the object or table.
false	A single ParameterInfo object is returned that provides information about the given parameter.	The whole sub-tree rooted at the given object or table path, this includes the object at the path itself. All objects must be included even if they are empty.

The result must include only parameters, objects, and tables that are actually implemented by the CPE. If a parameter is listed then a getParameterValue(String) method called with this parameter's path should succeed. As a convenience, the ParameterInfo class provides a getParameterValue() method as a short cut to the value.

For example, assume the following instances:

```
IGD.LAN.1.Hosts.
IGD.LAN.1.Hosts.HostNumberOfEntries
IGD.LAN.1.Hosts.Host.
IGD.LAN.1.Hosts.Host.1.
IGD.LAN.1.Hosts.Host.1.Active
IGD.LAN.1.Hosts.Host.2.
IGD.LAN.1.Hosts.Host.2.Active
IGD.LAN.2.Hosts.
IGD.LAN.2.Hosts.HostNumberOfEntries
```

The following table demonstrates some of the different results based on these example instances.

Table 131.9 *Example Get Parameter Names*

Parameter Name	Next level	Results	Comments
IGD.LAN.1.	false	IGD.LAN.1. IGD.LAN.1.Hosts. IGD.LAN.1.Hosts.HostNumberOfEntries IGD.LAN.1.Hosts.Host. IGD.LAN.1.Hosts.Host.1. IGD.LAN.1.Hosts.Host.1.Active IGD.LAN.1.Hosts.Host.2. IGD.LAN.1.Hosts.Host.2.Active	The path specifies an instance in at table and since the Next Level is false the whole sub-tree must be returned, including the root of the sub-tree.
	true	IGD.LAN.1.Hosts.	The path is the same, an instance in a table, but now only the children must be returned for the source. There is only one child, Hosts. This must be returned as an object path.
IGD.LAN.1.Hosts.« 1.Active	false	IGD.LAN.1.Hosts.Host. 1.Active	The path is a parameter path, therefore only the source is returned.
	true	Fault 9003 Invalid Arguments, next level must be false for a parameter path.	Next Level must not be set to true for a parameter path
IGD.LAN.1	false or true	Fault 9003 Invalid Arguments, it is not a parameter path but an instance id	It is not allowed to specify a parameter path that is actual pointing to an instance.

For example:

```
Collection<ParameterInfo> pinfos = connector.getParameterNames("Device.");
for ( ParameterInfo info : pinfos ) {
  if ( info.isParameter() ) {
     System.out.println(
         connector.getParameterValue(info.getName()).getValue());
  }
}
```

131.4.4 Add Object

The AddObject RPC creates a new instance in a table. There basic form for this RPC is to create an object and return the name of this object. It is also possible to specify an alias (a name specified in square brackets) after the table path. In that case, the alias is used as the node name. In either case, the path must be a valid table path pointing to a an existing MAP or LIST node.

When an object is added without an alias then the TR069 Connector must assign a unique id. TR-069 mandates that this id is unique for the table. The TR069 Connector must be able to create and maintain such a persistent id range. The Connector must ensure that any id chosen is not actually already in use or has been handed out recently. How such an id is calculated and maintained is implementation dependent.

If alias based addressing is used, a name between square brackets, then the alias is retrieved from the square brackets. The DMT must then be verified that no node exists in the corresponding table. If it does already exist, an INVALID_PARAMETER_NAME exception is thrown. Otherwise the alias is returned as the selected name.

If the corresponding MAP or LIST node has a Meta Node with a MIME type of application/x-tr-69-eager then the alias or instance id must be used to create the node. Otherwise the alias or instance id must be returned without creating the node. The purpose of this lazy creation is to allow a single Set Parameter Values RPC to atomically create a number of nodes and set their values.

For example:

```
String id = connector.addObject( "Starwars.CP.3.Obiwan.");
connector.setParameterValue( "Starwars.CP.3.Obiwan." + id+ ".Name",
                                            "cp30", TR069_STRING );
```

The previous code gets an assigned id with the addObject(String) method. The setParameterValue(String,String,int) then assigns the string cp30 to the Name node. This will first create the actual node since it was not created in the addObject(String) method and then sets the value of the DMT Starwars/CP/3/Obiwan/<id>/Name node.

The addObject(String) method requires an atomic session. If a non-atomic session is used then the addObject(String) method must not attempt to create any objects and an exception must be thrown.

131.4.5 Delete Object

The DeleteObject RPC deletes an object from the tree, it takes the instance path as argument. This behavior is implemented in the deleteObject(String) method. The corresponding node must be deleted if it exists. No error must be raised if the node does not exist in the DMT.

For example, deleting the object created in *Add Object* on page 660:

```
connector.deleteObject("Starwars.CP.3.Obiwan.cp30.");
```

131.5 Error and Fault Codes

The TR069 Connector must translate any Dmt Admin codes into a TR-069 fault code. Since the methods in the TR069Connector only relate to a single value it is possible to provide a mapping from Dmt Exception codes to TR-069 fault codes. It is the responsibility of the Protocol Adapter to aggregate these errors in the response to a SetParameterValues RPCs.

A TR069 Connector must prevent exceptions from happening and ensure that the different applicable error cases defined in the TR-069 RPCs are properly reported as a TR069 Exception with the intended fault code. However, this section defines a list of default translations between Dmt Exceptions and TR-069 fault codes.

The following table contains the exceptions and the resulting fault codes. Any obligations that are mandated by the TR-069 protocol are the responsibility of the TR-069 Protocol Adapter. The Dmt Exception is available from the TR-069 Exception for further inspection.

Table 131.10 Exceptions to TR-069 Fault code.

Exception	Fault code	Comments
ALERT_NOT_ROUTED	INTERNAL_ERROR	
COMMAND_FAILED	INTERNAL_ERROR	
COMMAND_NOT_ALLOWED	REQUEST_DENIED	
CONCURRENT_ACCESS	INTERNAL_ERROR	
DATA_STORE_FAILURE	INTERNAL_ERROR	
FEATURE_NOT_SUPPORTED	REQUEST_DENIED	
INVALID_URI	INVALID_PARAMETER_NAME	
LIMIT_EXCEEDED	RESOURCES_EXCEEDED	
METADATA_MISMATCH	INVALID_PARAMETER_TYPE	
NODE_ALREADY_EXISTS	INTERNAL_ERROR	
NODE_NOT_FOUND	INVALID_PARAMETER_NAME	
PERMISSION_DENIED	NON_WRITABLE_PARAMETER	
REMOTE_ERROR	INTERNAL_ERROR	
ROLLBACK_FAILED	INTERNAL_ERROR	
SESSION_CREATION_TIMEOUT	REQUEST_DENIED	
TRANSACTION_ERROR	REQUEST_DENIED	
UNAUTHORIZED	REQUEST_DENIED	
URI_TOO_LONG	INVALID_PARAMETER_NAME	
Dmt Illegal State Exception	INTERNAL_ERROR	
Security Exception	REQUEST_DENIED	
Other Exceptions	REQUEST_DENIED	

131.6 Managing the RMT

The RMT is not a native TR-069 model as it is not defined by BBF and it takes advantage of the Dmt Admin features. This section therefore shows a number of examples how the RMT can be managed from an ACS.

For example, on a specific CPE the following bundles are installed, the given name is the location

```
System Bundle
org-apache-felix-webconsole
org-apache-felix-configadmin
org-eclipse-equinox-scr
jp-co-ntt-admin
de-telekom-shell
```

The intention is to:

- Uninstall `org-apache-felix-configadmin`,
- Install and start `org-eclipse-equinox-cm`,

- Update jp-co-ntt-admin.

After the successful reconfiguration, the framework must restart. As framework changes must happen in a atomic session, the following parameters must be set in a single RPC:

```
SetParameterValues {
  Framework.Bundle.org-apache-felix-configadmin.RequestedState = UNINSTALLED
  Framework.Bundle.jp-co-ntt-admin.URL                         = http://....
  Framework.Bundle.org-eclipse-equinox-cm.URL                  = http://....
  Framework.Bundle.org-eclipse-equinox-cm.RequestedState       = ACTIVE
  Framework.Bundle.org-eclipse-equinox-cm.AutoStart            = true
  Framework.Bundle.Systemþ0020Bundle.URL                       = ""
}
```

The Protocol Adapter must open an atomic session on the $ node as defined in the RMT. It will then set all the parameters in the previous list. As the Framework/Bundle/org-eclipse-equinox-cm node does not exist, the TR069 Connector will create it because it is below a writable MAP node. The System Bundle is updated with an empty string, signalling an update. A System Bundle update is a framework restart.

Once the session is committed after all the SetParameterValues elements are executed the Data Plugin will perform the actions and report success or failure. The handler must then restart the framework after the commit has returned.

131.7 Native TR-069 Object Models

This section provides an example of a Data Plugin that provides a native TR-069 Object Model. As example is chosen a naive implementation of the Configuration Admin service. The object model implemented has the following definition:

Path	Type	Write	Read	Description
CM.{i}.	Object			
CM.{i}.Pid	string	x	x	The PID
CM.{i}.Properties.{i}.	Object			Property nodes
CM.{i}.Properties.{i}.Key	string	x	x	The key
CM.{i}.Properties.{i}.Value	string	x	x	Comma separated values

The corresponding DMT sub-tree is defined like:

Name	Act	Type	Card.	S	Description
CM	Get	MAP	1	P	Base node for the CM model
[string]	Get Set Add Del	Configuration	0..n	D	A MAP of the PID
InstanceId	Get	int	1	P	The persistent instance Id
Pid	Get	string	1	P	The PID of the configuration
Properties	Get	MAP	1	P	The properties
[string]	Get Set Add Del	LIST	0..n	D	A property definitions; a property consists of a list of strings. Single values are just a list with one element.
[index]	Get Set Add Del	string	0..n	D	An element in the list

The Protocol Adapter allows an ACS to access the data model implemented in the Dmt Plugin. It also allows the creation of new configuration objects.

131.8 org.osgi.service.tr069todmt

TR069 Connector Service Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.tr069todmt; version="[1.0,2.0)"`

Example import for providers of the API in this package:

`Import-Package: org.osgi.service.tr069todmt; version="[1.0,1.1)"`

131.8.1 Summary

- `ParameterInfo` - Maps to the TR-069 `ParameterInfoStruct` that is returned from the TR069Connector.getParameterNames(String, boolean) method.
- `ParameterValue` - Maps to the TR-069 `ParameterValueStruct`
- `TR069Connector` - A TR-069 Connector is an assistant to a TR-069 Protocol Adapter developer.
- `TR069ConnectorFactory` - A service that can create TR069 Connector
- `TR069Exception` - This exception is defined in terms of applicable TR-069 fault codes.

131.8.2 public interface ParameterInfo

Maps to the TR-069 `ParameterInfoStruct` that is returned from the TR069Connector.getParameterNames(String, boolean) method.

131.8.2.1 public ParameterValue getParameterValue() throws TR069Exception

□ Provide the value of the node. This method throws an exception if it is called for anything but a parameter

Returns The Parameter Value of the corresponding object

Throws `TR069Exception`– If there is a problem

131.8.2.2 public String getPath()

□ The path of the parameter, either a parameter path, an instance path, a table path, or an object path.

Returns The name of the parameter

131.8.2.3 public boolean isParameter()

□ Returns `true` of this is a parameter, if it returns `false` it is an object or table.

Returns `true` for a parameter, `false` otherwise

131.8.2.4 public boolean isWriteable()

□ Return `true` if this parameter is writeable, otherwise `false`. A parameter is writeable if the SetParamaterValue with the given name would be successful if an appropriate value was given. If this is a table path, the method specifies whether or not AddObject would be successful. If the parameter path points to a table instance, the method specifies whether or not DeleteObject would be successful.

Returns If this parameter is writeable

131.8.3 public interface ParameterValue

Maps to the TR-069 ParameterValueStruct

131.8.3.1 public String getPath()

□ This is the path of a Parameter. In TR-069 this is called the Parameter Name.

Returns The path of the parameter

131.8.3.2 public int getType()

□ The type of the parameter. One of TR069Connector.TR069_INT, TR069Connector.TR069_UNSIGNED_INT, TR069Connector.TR069_LONG, TR069Connector.TR069_UNSIGNED_LONG, TR069Connector.TR069_STRING, TR069Connector.TR069_DATETIME, TR069Connector.TR069_BASE64, TR069Connector.TR069_HEXBINARY. This method is not part of the ParameterValueStruct but is necessary to encode the type in the XML.

Returns The parameter type

131.8.3.3 public String getValue()

□ This is the value of the parameter. The returned value must be in a representation defined by the TR-069 protocol.

Returns The value of the parameter

131.8.4 public interface TR069Connector

A TR-069 Connector is an assistant to a TR-069 Protocol Adapter developer. The connector manages the low level details of converting the different TR-069 RPCs to a Device Management Tree managed by Dmt Admin. The connector manages the conversions from the TR-069 Object Names to a node in the DMT and vice versa.

The connector uses a Dmt Session from the caller, which is given when the connector is created. The connector does not implement the exact RPCs but only provides the basic functions to set and get the parameters of an object as well as adding and deleting an object in a table. A TR-069 developer must still parse the XML, handle the relative and absolute path issues, open a Dmt Session etc.

The connector assumes that each parameter or object path is relative to the root of the Dmt Session.

This connector must convert the TR-069 paths to Dmt Admin URIs. This conversion must take into account the LIST and MAP concepts defined in the specifications as well as the synthetic parameters NumberOfEntries and Alias. These concepts define the use of an InstanceId node that must be used by the connector to provide a TR-069 table view on the LIST and MAP nodes.

131.8.4.1 public static final String PREFIX = "application/x-tr-069-"

The MIME type prefix.

131.8.4.2 public static final int TR069_BASE64 = 64

Constant representing the TR-069 base64 type.

131.8.4.3 public static final int TR069_BOOLEAN = 32

Constant representing the TR-069 boolean type.

131.8.4.4 public static final int TR069_DATETIME = 256

Constant representing the TR-069 date time type.

131.8.4.5 public static final int TR069_DEFAULT = 0

Constant representing the default or unknown type. If this type is used a default conversion will take place

131.8.4.6 public static final int TR069_HEXBINARY = 128

Constant representing the TR-069 hex binary type.

131.8.4.7 public static final int TR069_INT = 1

Constant representing the TR-069 integer type.

131.8.4.8 public static final int TR069_LONG = 4

Constant representing the TR-069 long type.

131.8.4.9 public static final String TR069_MIME_BASE64 = "application/x-tr-069-base64"

Constant representing the TR-069 base64 type.

131.8.4.10 public static final String TR069_MIME_BOOLEAN = "application/x-tr-069-boolean"

Constant representing the TR-069 boolean type.

131.8.4.11 public static final String TR069_MIME_DATETIME = "application/x-tr-069-dateTime"

Constant representing the TR-069 date time type.

131.8.4.12 public static final String TR069_MIME_DEFAULT = "application/x-tr-069-default"

Constant representing the default or unknown type. If this type is used a default conversion will take place

131.8.4.13 public static final String TR069_MIME_EAGER = "application/x-tr-069-eager"

Constant representing the TR-069 eager type.

131.8.4.14 public static final String TR069_MIME_HEXBINARY = "application/x-tr-069-hexBinary"

Constant representing the TR-069 hex binary type.

131.8.4.15 public static final String TR069_MIME_INT = "application/x-tr-069-int"

Constant representing the TR-069 integer type.

131.8.4.16 public static final String TR069_MIME_LONG = "application/x-tr-069-long"

Constant representing the TR-069 long type.

131.8.4.17 public static final String TR069_MIME_STRING = "application/x-tr-069-string"

Constant representing the TR-069 string type.

131.8.4.18 public static final String TR069_MIME_STRING_LIST = "application/x-tr-069-string-list"

Constant representing the TR-069 string list type.

131.8.4.19 public static final String TR069_MIME_UNSIGNED_INT = "application/x-tr-069-unsignedInt"

Constant representing the TR-069 unsigned integer type.

131.8.4.20 public static final String TR069_MIME_UNSIGNED_LONG = "application/x-tr-069-unsignedLong"

Constant representing the TR-069 unsigned long type.

131.8.4.21 public static final int TR069_STRING = 16

Constant representing the TR-069 string type.

131.8.4.22 public static final int TR069_UNSIGNED_INT = 2

Constant representing the TR-069 unsigned integer type.

131.8.4.23 public static final int TR069_UNSIGNED_LONG = 8

Constant representing the TR-069 unsigned long type.

131.8.4.24 public String addObject(String path) throws TR069Exception

path A table path with an optional alias at the end

□ Add a new node to the Dmt Admin as defined by the AddObject RPC. The path must map to either a `LIST` or `MAP` node as no other nodes can accept new children.

If the path ends in an alias (`[ ALIAS ]`) then the node name must be the alias, however, no new node must be created. Otherwise, the Connector must calculate a unique instance id for the new node name that follows the TR-069 rules for instance ids. That is, this id must not be reused and must not be in use. That is, the id must be reserved persistently.

If the `LIST` or `MAP` node has a Meta Node with a MIME type application/x-tr-069-eager then the node must be immediately created. Otherwise no new node must be created, this node must be created when the node is accessed in a subsequent RPC.

The alias name or instance id must be returned as identifier for the ACS.

Returns The name of the new node.

Throws `TR069Exception`– The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005. If an AddObject request would result in exceeding the maximum number of such objects supported by the CPE, the CPE MUST return a fault response with the Resources Exceeded (9004) fault code.

131.8.4.25 public void close()

□ Close this connector. This will **not** close the corresponding session.

131.8.4.26 public void deleteObject(String objectPath) throws TR069Exception

objectPath The path to an object in a table to be deleted.

□ Delete an object from a table. A missing node must be ignored.

Throws `TR069Exception`– The following fault codes are defined for this method: 9001, 9002, 9003, 9005. If the fault is caused by an invalid objectPath value, the Invalid Parameter Name fault code (9005) must be used instead of the more general Invalid Arguments fault code (9003). A missing node for `objectPath` must be ignored.

131.8.4.27 public Collection<ParameterInfo> getParameterNames(String objectOrTablePath, boolean nextLevel) throws TR069Exception

objectOrTablePath A path to an object or table.

nextLevel If `true` consider only the children of the object or table addressed by `path`, otherwise include the whole sub-tree, including the addressed object or table.

□ Getting the ParameterInfo objects addressed by path. This method is intended to be used to implement the GetParameterNames RPC.

The connector must attempt to create any missing nodes that are needed for the `objectOrTablePath` by using the toURI(String, boolean) method with `true`.

This method must traverse the sub-tree addressed by the path and return the paths to all the objects, tables, and parameters in that tree. If the nextLevel argument is `true` then only the children object, table, and parameter information must be returned.

The returned ParameterInfo objects must be usable to discover the sub-tree.

If the child nodes have an InstanceId node then the returned names must include the InstanceId values instead of the node names.

If the parent node is a MAP, then the synthetic Alias parameter must be included.

Any MAP and LIST node must include a ParameterInfo for the corresponding NumberOfEntries parameter.

Returns A collection of ParameterInfo objects representing the resulting child parameter, objects, and tables as defined by the TR-069 ParameterInfoStruct.

Throws TR069Exception– If the fault is caused by an invalid ParameterPath value, the Invalid Parameter Name fault code (9005) MUST be used instead of the more general Invalid Arguments fault code (9003). A ParameterPath value must be considered invalid if it is not an empty string and does not exactly match a parameter or object name currently present in the data model. If nextLevel is true and objectOrTablePath is a parameter path rather than an object/table path, the method must return a fault response with the Invalid Arguments fault code (9003). If the value cannot be gotten for some reason, this method can generate the following fault codes::

- 9001 TR069Exception.REQUEST_DENIED
- 9002 TR069Exception.INTERNAL_ERROR
- 9003 TR069Exception.INVALID_ARGUMENTS
- 9005 TR069Exception.INVALID_PARAMETER_NAME

131.8.4.28 public ParameterValue getParameterValue(String parameterPath) throws TR069Exception

parameterPath A parameter path (must refer to a valid parameter, not an object or table).

□ Getting a parameter value. This method should be used to implement the GetParameterValues RPC. This method does **not** handle retrieving multiple values as the corresponding RPC can request with an object or table path, this method only accepts a parameter path. Retrieving multiple values can be achieved with the getParameterNames(String, boolean).

If the parameterPath ends in NumberOfEntries then the method must synthesize the value. The parameterPath then has a pattern like (object-path)(table-name)NumberOfEntries. The returned value must be an TR069_UNSIGNED_INT that contains the number of child nodes in the table (object-path)(table-name). For example, if A.B.CNumberOfEntries is requested the return value must be the number of child nodes under A/B/C.

If the value of a an Alias node is requested then the name of the parent node must be returned. For example, if the path is M.X.Alias then the returned value must be X.

The connector must attempt to create any missing nodes along the way, creating parent nodes on demand.

Returns The name, value, and type triad of the requested parameter as defined by the TR-069 ParameterValueStruct.

Throws TR069Exception– The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005.

- 9001 TR069Exception.REQUEST_DENIED
- 9002 TR069Exception.INTERNAL_ERROR
- 9003 TR069Exception.INVALID_ARGUMENTS
- 9004 TR069Exception.RESOURCES_EXCEEDED
- 9005 TR069Exception.INVALID_PARAMETER_NAME

131.8.4.29 public void setParameterValue(String parameterPath, String value, int type) throws TR069Exception

parameterPath The parameter path

value A trimmed string value that has the given type. The value can be in either canonical or lexical representation by TR069.

type The type of the parameter (TR069_INT, TR069_UNSIGNED_INT,TR069_LONG, TR069_UNSIGNED_LONG,TR069_STRING, TR069_DATETIME,TR069_BASE64, TR069_HEXBINARY, TR069_BOOLEAN)

□ Setting a parameter. This method should be used to provide the SetParameterValues RPC. This method must convert the parameter Name to a URI and replace the DMT node at that place. It must follow the type conversions as described in the specification.

The connector must attempt to create any missing nodes along the way, creating parent nodes on demand.

If the value of a an Alias node is set then the parent node must be renamed. For example, if the value of M/X/Alias is set to Y then the node will have a URI of M/Y/Alias. The value must not be escaped as the connector will escape it.

Throws TR069Exception– The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005, 9006, 9007, 9008.

- 9001 TR069Exception.REQUEST_DENIED
- 9002 TR069Exception.INTERNAL_ERROR
- 9003 TR069Exception.INVALID_ARGUMENTS
- 9004 TR069Exception.RESOURCES_EXCEEDED
- 9005 TR069Exception.INVALID_PARAMETER_NAME
- 9006 TR069Exception.INVALID_PARAMETER_TYPE
- 9007 TR069Exception.INVALID_PARAMETER_VALUE
- 9008 TR069Exception.NON_WRITABLE_PARAMETER

131.8.4.30 public String toPath(String uri) throws TR069Exception

uri A Dmt Session relative URI

□ Convert a Dmt Session relative Dmt Admin URI to a valid TR-069 path, either a table, object, or parameter path depending on the structure of the DMT. The translation takes into account the special meaning LIST, MAP , Alias, and InstanceId nodes.

Returns An object, table, or parameter path

Throws TR069Exception– If there is an error

131.8.4.31 public String toURI(String name, boolean create) throws TR069Exception

name A TR-069 path

create If true, create missing nodes when they reside under a MAP or LIST

□ Convert a TR-069 path to a Dmt Session relative Dmt Admin URI. The translation takes into account the special meaning LIST, MAP, InstanceId node semantics.

The synthetic Alias or NumberOfEntries parameter cannot be mapped and must throw an TR069Exception.INVALID_PARAMETER_NAME.

The returned path is properly escaped for TR-069.

The mapping from the path to a URI requires support from the meta data in the DMT, it is not possible to use a mapping solely based on string replacements. The translation takes into account the semantics of the MAP and LIST nodes. If at a certain point a node under a MAP node does not exist then the Connector can create it if the create flag is set to true. Otherwise a non-existent node will terminate the mapping.

Returns A relative Dmt Admin URI

Throws `TR069Exception`– If there is an error

131.8.5 public interface TR069ConnectorFactory

A service that can create TR069 Connector

131.8.5.1 public TR069Connector create(DmtSession session)

session The session to use for the adaption. This session must not be closed before the TR069 Connector is closed.

□ Create a TR069 connector based on the given session .

The session must be an atomic session when objects are added and/or parameters are going to be set, otherwise it can be a read only or exclusive session. Due to the lazy creation nature of the TR069 Connector it is possible that a node must be created in a read method after a node has been added, it is therefore necessary to always provide an atomic session when an ACS session requires modifying parameters.

Returns A new TR069 Connector bound to the given session

131.8.6 public class TR069Exception extends RuntimeException

This exception is defined in terms of applicable TR-069 fault codes. The TR-069 specification defines the fault codes that can occur in different situations.

131.8.6.1 public static final int INTERNAL_ERROR = 9002

9002 Internal error

131.8.6.2 public static final int INVALID_ARGUMENTS = 9003

9003 Invalid Arguments

131.8.6.3 public static final int INVALID_PARAMETER_NAME = 9005

9005 Invalid parameter name (associated with Set/GetParameterValues, GetParameterNames, Set/GetParameterAttributes, AddObject, and DeleteObject)

131.8.6.4 public static final int INVALID_PARAMETER_TYPE = 9006

9006 Invalid parameter type (associated with SetParameterValues)

131.8.6.5 public static final int INVALID_PARAMETER_VALUE = 9007

9007 Invalid parameter value (associated with SetParameterValues)

131.8.6.6 public static final int METHOD_NOT_SUPPORTED = 9000

9000 Method not supported

131.8.6.7 public static final int NON_WRITABLE_PARAMETER = 9008

9008 Attempt to set a non-writable parameter (associated with SetParameterValues)

131.8.6.8 public static final int NOTIFICATION_REJECTED = 9009

9009 Notification request rejected (associated with SetParameterAttributes method).

131.8.6.9 public static final int REQUEST_DENIED = 9001

9001 Request denied (no reason specified

131.8.6.10 public static final int RESOURCES_EXCEEDED = 9004

9004 Resources exceeded (when used in association with SetParameterValues, this MUST NOT be used to indicate parameters in error)

131.8.6.11 public TR069Exception(String message)

message The message

□ A default constructor when only a message is known. This will generate a INTERNAL_ERROR fault.

131.8.6.12 public TR069Exception(String message, int faultCode, DmtException e)

message The message

faultCode The TR-069 defined fault code

e

□ A Constructor with a message and a fault code.

131.8.6.13 public TR069Exception(String message, int faultCode)

message The message

faultCode The TR-069 defined fault code

□ A Constructor with a message and a fault code.

131.8.6.14 public TR069Exception(DmtException e)

e The Dmt Exception

□ Create a TR069Exception from a Dmt Exception.

131.8.6.15 public DmtException getDmtException()

Returns the corresponding Dmt Exception

131.8.6.16 public int getFaultCode()

□ Answer the associated TR-069 fault code.

Returns Answer the associated TR-069 fault code.

131.9 References

[1] *TR-069 Amendment 3*
https://www.broadband-forum.org/technical/download/TR-069_Amendment-3.pdf

[2] *TR-106 Amendment 3*
https://www.broadband-forum.org/technical/download/TR-106_Amendment-3.pdf

[3] *XML Schema Part 2: Datatypes Second Edition*
https://www.w3.org/TR/xmlschema-2/

[4] *SOAP 1.1*
https://www.w3.org/TR/soap11/

[5] *Extensible Markup Language (XML) 1.0 (Second Edition)*
https://www.w3.org/TR/xml/#NT-Letter

[6] *Broadband Forum*
https://www.broadband-forum.org/

132 Repository Service Specification

Version 1.1

132.1 Introduction

The guiding force behind the OSGi Specifications is a reusable component model. The *OSGi Core Release 8* provides a solid foundation for such a component model by providing a component collaboration framework with a comprehensive management model. The service specifications provide the abstract APIs to allow many different collaborations between components. This Repository Service Specification provides the capability to manage the external access to components and other resources.

Though the Repository service can be used as a standalone service to search and retrieve general binary artifacts, called resources, it is intended to be used in conjunction with the [6] *Resolver Service Specification*.

The model of the Repository is based on the generic Requirement-Capability model defined in [3] *Resource API Specification*, this chapter relies on the definitions of the generic model.

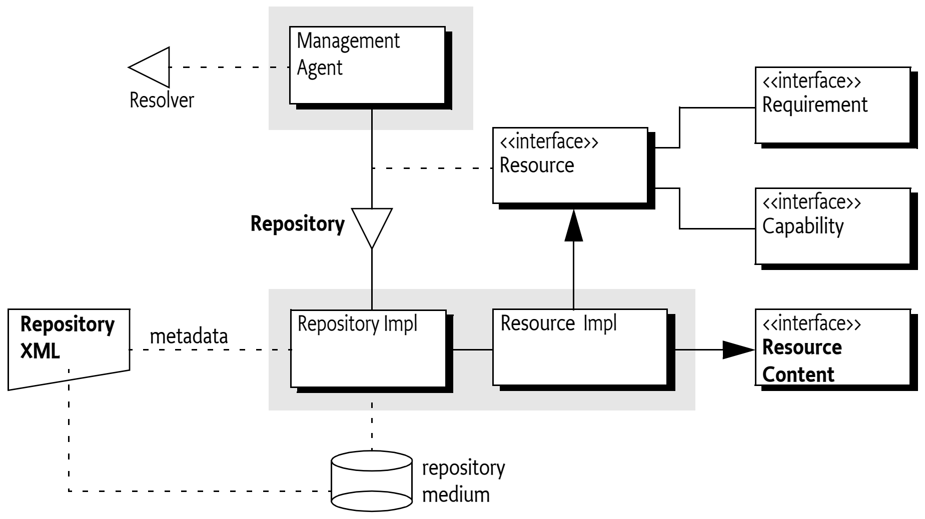
132.1.1 Essentials

- *External* - Provide access to external components and resources.
- *Resolve* - The Repository API must be closely aligned with the Resolver API since they are intended to be used in conjunction.
- *Searching* - Support general queries.
- *Metadata* - Allow resources to provide content information.
- *Retrieval* - Allow the retrieval of Resources from remote locations.
- *Batching* - Repositories must be able to batch queries.
- *Distribution* - Allow Repositories to be defined with a simple storage scheme such that Repositories can be distributed on a removable media like a CD/DVD.
- *Mirroring* - Repositories must be able to support selecting a remote site based on the local situation.

132.1.2 Entities

- *Repository* - A facade to a (remote) set of resources described by capabilities.
- *Resource* - An artifact that has requirements that must be satisfied before it is available but provides capabilities when it becomes available.
- *Requirement* - An expression that asserts a capability.
- *Capability* - Describes a feature of the resource so that it can be required by a requirement.
- *Resource Content* - Provides access to the underlying bytes of the resource in the default format.

Figure 132.1 Class and Service overview



132.1.3 Synopsis

There are many different repositories available on the Internet or on fixed media. A repository can be made available to bundles by providing a Repository service. If such a bundle, for example a Management Agent performing a provisioning operation, finds that it has an unmatched requirement then it can query the repository services to find matching capabilities. The Repository service can implement the query in many different ways. It can ship the requirement to a remote side to be processed or it can process the query locally.

This specification also provides an XML schema that can be used to describe a Repository. Instances of this schema can be downloaded from a remote repository for local indexing or they can be stored for example on a DVD together with the resources.

132.2 Using a Repository

The Repository service provides an abstraction to a, potentially remote, set of resources. In the generic Capability-Requirement model, resources are modeled to declare capabilities and requirements. The primary purpose of a Repository is to enable a management agent that uses the Resolver API to leverage a wide array of repositories. This Repository service specification allows different Repository providers to be installed as bundles, and each bundle can register multiple Repository services. The Repository is sufficiently abstract to allow many different implementations.

Repository services are identified by a number of service properties:

- `service.pid` - A mandatory unique identity for this Repository service.
- `service.description` - An optional human readable name for this Repository.
- `repository.url` - Optional URLs to landing pages of the repository, if they exist.

In general, the users of the Repository service should aggregate all services in the service registry. This strategy allows the deployer to control the available Repositories. The following example, using Declarative Service annotations to show the dependencies on the service registry, shows how to aggregate the different Repository services.

```
List<Repository> repos = new CopyOnWriteArrayList<Repository>();
```

```
@Reference(
cardinality = ReferenceCardinality.MULTIPLE,
policy = ReferencePolicy.DYNAMIC)
void addRepository( Repository repo )    { repos.add(repo); }
void removeRepository( Repository repo ) { repos.remove(repo); }
```

To access a resource in a Repository service it is necessary to construct a requirement, pass this to the Repository service, and then use the returned capabilities to satisfy the resolver or to get the resource from the capability. The Repository then returns all matching capabilities. The requirement matches the capability if their namespaces match and the requirement's filter is absent or matches the attributes.

The findProviders(Collection) method takes a Collection of requirements. The reason for this collection is that it allows the caller to specify multiple requirements simultaneously so that Repositories can batch requests, the requirements in this collection are further unrelated. That is, they do not form an expression in any way. Multiple requirements as the parameter means that the result must be a map so that the caller can find out what requirement matched what capabilities. For example:

```
List<Capability> find( Requirement r ){
  List<Capability> result = new ArrayList<Capability>();

  for ( Repository repo : repos ) {
    Map<Requirement,Collection<Capability>> answer =
        repo.findProviders( Collections.singleton( r ) );
     result.addAll( answer.get( r ) );
  }
  return result;
}
```

Access to resources is indirect since the Repository returns capabilities. Each capability is declared in a resource and the getResource() method provides access to the underlying resource. Since each resource declares an osgi.identity capability it is possible to retrieve a resource from a repository if the identity name, type, and version are known. For example, to get a bundle resource:

```
Resource getResource( String type, String name, Version version ) {
  String filter = String.format(
    "(&(type=%s)(osgi.identity=%s)(version=%s))",
    type,
    name,
    version );

  RequirementBuilder builder = repo.newRequirementBuilder("osgi.identity");
  builder.addDirective("filter", filter);
  Requirement r = builder.build();

  List<Capability> capabilities = find( r );
  if ( capabilities.isEmpty() )
    return null;
  return capabilities.get( 0 ).getResource();
}
```

Resources that originate from Repository services must implement the RepositoryContent interface, this interface provides stream access to the default storage format. It is therefore possible to get the content with the following code.

```
InputStream getContent( String type, String name, Version version ) {
  Resource r = getResource( type, name, version );
  if ( r == null )
    return null;
  return ((RepositoryContent)r).getContent();
}
```

The getContent() method returns an Input Stream in the default format for that resource type. Resources from a Repository should also have one or more osgi.content capabilities that advertise the same resource in the same or different formats. The osgi.content capability has a number of attributes that provide information about the resource's download format:

- osgi.content - A unique SHA-256 for the content as read from the URL.
- url - A URL to the content.
- mime - An IANA MIME type for the content.
- size - Size in bytes of the content.

It is therefore possible to search for a specific MIME type and download that format. For example:

```
String getURL( String type, String name, Version version, String mime )
    throws Exception {
    Resource r = getResource( type, name, version );
    for ( Capability cap : r.getCapabilities( "osgi.content") ) {
        Map<String,Object> attrs = cap.getAttributes();
        String actual = (String) attrs.get("mime");
        if ( actual!=null && mime.equalsIgnoreCase( actual) ) {
            String url = (String) attrs.get( "url" );
            if ( url != null )
                return url;
        }
   }
   return null;
}
```

Since the osgi.content capability contains the SHA-256 digest as the osgi.content attribute it is possible to verify the download that it was correct.

Every resource has an osgi.identity capability. This namespace defines, in [2] *Framework Namespaces*, the possibility to add related resources, for example *javadoc* or *sources*. A resource then has informational requirements to osgi.identity capabilities; these requirements are marked with a classifier directive that holds the type of *relation*. The following example shows how it would be possible to find such a related resource:

```
InputStream getRelated(Resource resource,String classifier)
    throws Exception {
    for ( Requirement r : resource.getRequirements( "osgi.identity") ) {
        if ( classifier.equals( r.getDirectives().get( "classifier") ) ) {
            Collection<Capability> capabilities =
                 repository.findProviders( Collections.singleton( r )).get( r );

             if ( capabilities.isEmpty())
                continue;

            Capability c = capabilities.iterator().next();
            Resource related = c.getResource();
            return ((RepositoryContent)related).getContent();
        }
    }
    return null;
}
```

132.2.1 Combining Requirements

In some cases it may be useful to find resources in the repository that satisfy criteria across multiple namespaces.

A simple Requirement object can contain a filter that makes assertions about capability attributes within a single namespace. So for example, a single requirement can state that a package org.example.mypkg must be exported in a version between 3.1 inclusive and 4.0 exclusive:

```
RequirementBuilder rb = repo.newRequirementBuilder("osgi.wiring.package");
String rf = "(&(osgi.wiring.package=org.example.mypkg)"
            + "(version>=3.1)(!(version>=4.0)))";
rb.addDirective("filter", rf);
Requirement r = rb.build();
```

This requirement contains three conditions on the osgi.wiring.package capability.

In some situations it may be needed to specify requirements that cover multiple namespaces. For example a bundle might be needed that exports the above package, but the bundle must also have the Apache License, Version 2.0 license. A resource's license is available as an attribute on the osgi.identity namespace. Constructing a constraint that combines requirements from multiple namespaces can be done by using an Expression Combiner, which can be obtained from the Repository service. The Repository service provides a findProviders(RequirementExpression) overload that can take a requirement expression and returns a Promise to a collection of matching resources.

```
RequirementBuilder lb = repo.newRequirementBuilder("osgi.identity");
String lf = "(license=http://opensource.org/licenses/Apache-2.0)";
lb.addDirective("filter", lf);

RequirementExpression expr = repo.getExpressionCombiner().and(
  lb.buildExpression(), rb.buildExpression());

Promise<Collection<Resource>> p = repo.findProviders(expr);

// Let findProviders() do its work async and update a ui component
// once the result is available
p.then(new Success<Collection<Resource>, Void>() {
  public Promise<Void> call(Promise<Collection<Resource>> resolved)
      throws Exception {
    ui.update(resolved.getValue());
    return null;
  }
});

// Instead of the async chain above its also possiblye to
// wait for the promise value synchronously:
//   Collection<Resource> resources = p.getValue();
```

For more details on OSGi Promises, see the *Promises Specification* on page 1389.

132.3 Repository

A Repository service provides access to capabilities that satisfy a given requirement. A Repository can be the facade of a remote server containing a large amount of resources, a repository on removable media, or even a collection of bundles inside a ZIP file. A Repository communicates in terms of requirements and capabilities as defined in [3] *Resource API Specification*. This model is closely aligned with the [6] *Resolver Service Specification*.

A Repository service must be registered with the service properties given in the following table.

Table 132.1 *Repository Service Properties*

Attribute	Opt	Type	Description
service.pid	mandatory	String	A globally unique identifier for this Repository.
service.description	optional	String	The Repository Name
repository.url	optional	String+	URLs related to this Repository.

The Repository implements the following methods:

- findProviders(Collection) - For each requirement find all the capabilities that match that requirement and return them as a Map<Requirement,Collection<Capability>>.
- findProviders(RequirementExpression) - Find all resources that match the requirement expression. The requirement expression is used to combine multiple requirements using the and, or and not operators.
- getExpressionCombiner() - Obtain an expression combiner. This expression combiner is used to produce requirement expressions from simple requirements or other requirement expressions.
- newRequirementBuilder(String) - Obtain a convenience builder for Requirement objects.

A Repository must not perform any namespace specific actions or matching. The Repository must therefore match a requirement to a capability with the following rules:

- The namespace must be identical, and
- The requirement's filter is absent or it must match the capability's attributes.

Resources originating from a Repository service must additionally:

- Implement the RepositoryContent interfaces, see *Repository Content* on page 676.
- Provide at least one osgi.content Capability, see *osgi.content Namespace* on page 676.

132.3.1 Repository Content

Resources originating from a Repository must implement the RepositoryContent interface. The purpose of this interface is to allow users of the Repositories access to an Input Stream that provides access to the resource.

The RepositoryContent interface provides a single method:

- getContent() - Return an Input Stream for the resource, if more than one osgi.content capability is present the content associated with the first capability is returned.

132.4 osgi.content Namespace

A resource is a logical concept, to install a resource in an environment it is necessary to get access to its *contents*. A resource can be formatted in different ways. It is possible to deliver a bundle as a JAR file, a Pack200 file, or some other format. In general, the RepositoryContent interface provides access to the default format.

The Repository can advertise the different formats with osgi.content capabilities. Each of those capabilities is identified with a unique SHA-256 checksum and has a URL for the resource in the specified format. The size and mime attributes provide information the download format, this can be used for selection. If more than one osgi.content capability is associated with a resource, the first capability must represent the default format. If the resource has a standard or widely used format (e.g., JAR for bundles and ESA for subsystems), and that format is provided as part of the repository, then that format should be the default format.

The osgi.content Namespace supports the attributes defined in the following table and ContentNamespace.

Table 132.2 osgi.content definition

Name	Kind	M/O	Type	Syntax	Description
osgi.content	CA	M	String	[0-9a-fA-F]{64}	The SHA-256 hex encoded digest for this resource
url	CA	M	String	<url>	The URL to the bytes. This must be an absolute URL.
size	CA	M	Long	[0-9]+	The size of the resource in bytes as it will be read from the URL.
mime	CA	M	String	<mime type>	An IANA defined MIME type for the format of this content.

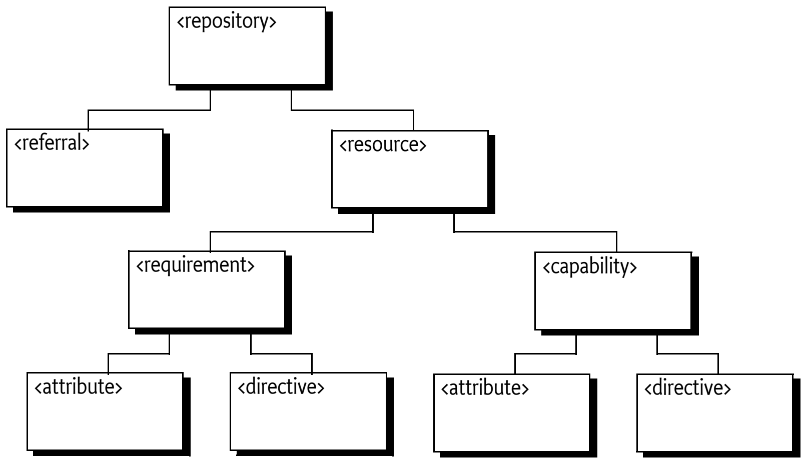
132.5 XML Repository Format

This is an optional part of the specification since the Repository interface does not provide access how the Repository obtains its information. However, the purpose of this part of the specification is to provide a commonly recognized format for interchanging Repository metadata.

This section therefore describes an XML schema to represent Repository content. It is expected that Internet based Repositories can provide such an XML file to clients. A Repository XML file can be used as a common interchange format between multiple Repository implementations.

The Repository XML describes a number of resources with their capabilities and requirements. Additionally the XML can refer to other Repository XML files. The XML Schema can be found at its XML namespace, see *XML Repository Schema* on page 681. The XML structure, which closely follows the Requirement-Capability model, is depicted in Figure 132.2.

Figure 132.2 XML Structure



The different elements are discussed in the following sections. All types are derived from the XML Schema types, see [4] *XML Schema Part 2: Data types Second Edition*. Any relative URIs in a Repository XML file must be resolved as specified in [5] *XML Base (Second Edition), Resolving Relative URIs*.

132.5.1 Repository Element

The repository element is the root of the document. The repository element has the following child elements:

- referral* - Referrals to other repositories for a federated model, see *Referral Element* on page 678.
- resource* - Resource definitions, see *Resource Element* on page 678.

The repository element has the attributes defined in the following table.

Table 132.3 repository element attributes

Attribute	Type	Description
name	NCName	The name of this Repository. For informational purposes.
increment	long	Counter which increments every time the repository is changed. Can be used by clients to check for changes. The counter is not required to increase monotonically.

132.5.2 Referral Element

The purpose of the referral element is to allow a Repository to refer to other Repositories, allowing for federated Repositories. Referrals are applied recursively. However, this is not always desired. It is therefore possible to limit the depth of referrals. If the depth attribute is >= 1, the referred repository must be included but it must not follow any referrals from the referred repository. If the depth attribute is more than one, referrals must be included up to the given depth. Depths of referred repositories must also be obeyed, where referred repositories may reduce the effective depth but not increase it. For example if a top repository specifies a depth of 5 and a level 3 repository has a depth of 1 then the repository on level 5 must not be used. If not specified then there is no limit to the depth. Referrals that have cycles must be ignored, a resource of a given Repository must only occur once in a Repository.

The referral element has the attributes defined in the following table.

Table 132.4 referral element attributes

Attribute	Type	Description
depth	int	The max depth of referrals
url	anyURI	A URL to where the referred repository XML can be found. The URL can be absolute or relative to the URI of the current XML resource.

132.5.3 Resource Element

The resource element defines a Resource. The resource element has the following child elements:

- requirement* - The requirements of this resource, see *Requirement Element* on page 679.
- capability* - The capabilities of this resource, see *Capability Element* on page 678.

The Resource element has no attributes.

132.5.4 Capability Element

The capability element maps to a capability, it holds the attributes and directives. The capability element has the following child elements:

- directive* - The directives for the capability, see *Directive Element* on page 680.
- attribute* - The attributes for the capability, see *Attribute Element* on page 679.

The capability element has the attributes defined in the following table.

Table 132.5 *capability element attributes*

Attribute	Type	Description
namespace	token	The namespace of this capability

132.5.5 Requirement Element

The requirement element maps to a requirement, it holds the attributes and directives. The requirement element has the following child elements:

- directive* - The directives for the requirement, see *Directive Element* on page 680.
- attribute* - The attributes for the requirement, see *Attribute Element* on page 679.

The requirement element has the attributes defined in the following table.

Table 132.6 *requirement element attributes*

Attribute	Type	Description
namespace	token	The namespace of this requirement

132.5.6 Attribute Element

An attribute element describes an attribute of a capability or requirement. Attributes are used to convey information about the Capability-Requirement. Attributes for the capability are used for matching the requirement's filter. The meaning of attributes is described with the documentation of the namespace in which they reside.

Attributes are optionally typed according to the [1] *Framework Module Layer* specification. The default type is String, the value of the value attribute. However, if a type attribute is specified and it is not String then the value attribute must be converted according to the type attribute specifier. The syntax of the type attribute is as follows:

```
type    ::= list | scalar
list    ::= 'List<' scalar '>'    // no spaces between terminals
scalar  ::= 'String' | 'Version' | 'Long' | 'Double'
```

A list conversion requires the value to be broken in tokens separated by comma (',' \u002C). Whitespace around the list and around commas must be trimmed for non-String types. Each token must then be converted to the given type according to the scalar type specifier. The exact rules for the comma separated lists are defined in [1] *Framework Module Layer*, see *Bundle Capability Attributes*.

The conversion of value s, when scalar, must take place with the following methods:

- String - No conversion, use s
- Version - Version.parseVersion(s)
- Long - After trimming whitespace, Long.parseLong(s)
- Double - After trimming whitespace, Double.parseDouble(s)

The attribute element has the attributes defined in the following table.

Table 132.7 *attribute element attributes*

Attribute	Type	Description
name	token	The name of the attribute

Attribute	Type	Description
value	string	The value of the attribute.
type		The type of the attribute, the syntax is outlined in the previous paragraphs.

132.5.7 Directive Element

A directive element describes a directive of a capability or a requirement. Directives are used to convey information about the Capability-Requirement. The meaning of directives is described with the documentation of the namespace in which they reside.

The directive element has the attributes defined in the following table.

Table 132.8 *directive element attributes*

Attribute	Type	Description
name	token	The name of the attribute
value	string	The value of the attribute.

132.5.8 Sample XML File

The following example shows a very small XML file. The file contains one resource.

```
<repository name='OSGiRepository'
           increment='13582741'
           xmlns='http://www.osgi.org/xmlns/repository/v1.0.0'>
  <resource>

    <requirement namespace='osgi.wiring.package'>
      <directive name='filter' value=
                        '(&amp;(osgi.wiring.package=org.apache.commons.pool)(version&gt;=1.5.6))'/>
    </requirement>

    <requirement namespace='osgi.identity'>
      <directive name='effective' value='meta'/>
      <directive name='resolution' value='optional'/>
      <directive name='filter' value=
             '(&(version=1.5.6)(osgi.identity=org.acme.pool-src))'
        <directive name='classifier' value='sources'/>
    </requirement>

    <capability namespace='osgi.identity'>
      <attribute name='osgi.identity' value='org.acme.pool'/>
      <attribute name='version'type='Version' value='1.5.6'/>
      <attribute name='type' value='osgi.bundle'/>
    </capability>

    <capability namespace='osgi.content'>
      <attribute name='osgi.content' value='e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'
      <attribute name='url' value='http://www.acme.com/repository/org/acme/pool/org.acme.pool-1.5.6.jar'/>
      <attribute name='size' type='Long' value='4405'/>
      <attribute name='mime' value='application/vnd.osgi.bundle'/>
    </capability>

    <capability namespace='osgi.wiring.bundle'>
      <attribute name='osgi.wiring.bundle' value='org.acme.pool'/>
      <attribute name='bundle-version' type='Version' value='1.5.6'/>
    </capability>

    <capability namespace='osgi.wiring.package'>
      <attribute name='osgi.wiring.package' value='org.acme.pool'/>
      <attribute name='version' type='Version' value='1.1.2'/>
      <attribute name='bundle-version' type='Version' value='1.5.6'/>
      <attribute name='bundle-symbolic-name' value='org.acme.pool'/>
      <directive name='uses' value='org.acme.pool,org.acme.util'/>
    </capability>

  </resource>
</repository>
```

132.6 XML Repository Schema

The namespace of this schema is:

`http://www.osgi.org/xmlns/repository/v1.0.0`

The schema for this namespace can be found at the location implied in its name. The recommended prefix for this namespace is `repo`.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:repo="http://www.osgi.org/xmlns/repository/v1.0.0"
    targetNamespace="http://www.osgi.org/xmlns/repository/v1.0.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.0.1">

    <element name="repository" type="repo:Trepository" />
    <complexType name="Trepository">
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="resource" type="repo:Tresource" />
                <element name="referral" type="repo:Treferral" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The name of the repository. The name may contain
                    spaces and punctuation.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="increment" type="long">
            <annotation>
                <documentation xml:lang="en">
                    An indication of when the repository was last changed. Client's can
                    check if a
                    repository has been updated by checking this increment value.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tresource">
        <annotation>
            <documentation xml:lang="en">
                Describes a general resource with
                requirements and capabilities.
            </documentation>
        </annotation>
        <sequence>
            <element name="requirement" type="repo:Trequirement" minOccurs="0" maxOccurs="unbounded"/>
            <element name="capability" type="repo:Tcapability" minOccurs="1" maxOccurs="unbounded"/>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Treferral">
        <annotation>
            <documentation xml:lang="en">
                A referral points to another repository XML file. The
                purpose of this element is to create a federation of
                repositories that can be accessed as a single
                repository.
            </documentation>
        </annotation>
        <attribute name="depth" type="int" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The depth of referrals this repository acknowledges.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="url" type="anyURI" use="required">
            <annotation>
                <documentation xml:lang="en">
                    The URL to the referred repository. The URL can be
                    absolute or relative from the given repository's
                    URL.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tcapability">
        <annotation>
            <documentation xml:lang="en">
                A named set of type attributes and directives. A capability can be
                used to resolve a requirement if the resource is included.
            </documentation>
        </annotation>
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="directive" type="repo:Tdirective" />
                <element name="attribute" type="repo:Tattribute" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
                <documentation xml:lang="en">
                    Name space of the capability. Only requirements with the
                    same name space must be able to match this capability.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Trequirement">
        <annotation>
            <documentation xml:lang="en">
                A filter on a named set of capability attributes.
            </documentation>
        </annotation>
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="directive" type="repo:Tdirective" />
                <element name="attribute" type="repo:Tattribute" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
                <documentation xml:lang="en">
                    Name space of the requirement. Only capabilities within the
                    same name space must be able to match this requirement.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tattribute">
        <annotation>
            <documentation xml:lang="en">
                A named value with an optional type that decorates
                a requirement or capability.
            </documentation>
        </annotation>
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The name of the attribute.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="value" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The value of the attribute.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="type" type="repo:TpropertyType" default="String">
            <annotation>
                <documentation xml:lang="en">
                    The type of the attribute.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tdirective">
        <annotation>
            <documentation xml:lang="en">
                A named value of type string that instructs a resolver
                how to process a requirement or capability.
            </documentation>
        </annotation>
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The name of the directive.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="value" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The value of the directive.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <simpleType name="TpropertyType">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Version" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="List&lt;String&gt;" />
            <enumeration value="List&lt;Version&gt;" />
            <enumeration value="List&lt;Long&gt;" />
            <enumeration value="List&lt;Double&gt;" />
        </restriction>
    </simpleType>
    <attribute name="must-understand" type="boolean" default="false">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents to require that
                the document consumer understand the extension. This attribute must be
                qualified when used.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

132.7 Capabilities

Implementations of the Repository Service specification must provide the capabilities listed in this section.

132.7.1 osgi.implementation Capability

The Repository Service implementation bundle must provide the osgi.implementation capability with name osgi.repository. This capability can be used by provisioning tools and during resolution to ensure that a Repository Service implementation is present. The capability must also declare a uses constraint for the org.osgi.service.repository package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
         osgi.implementation="osgi.repository";
         uses:="org.osgi.service.repository";
         version:Version="1.1"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

132.7.2 osgi.service Capability

The Repository Service implementation must provide a capability in the osgi.service namespace representing the Repository service. This capability must also declare a uses constraint for the org.osgi.service.repository package. For example:

```
Provide-Capability: osgi.service;
         objectClass:List<String>="org.osgi.service.repository.Repository";
         uses:="org.osgi.service.repository"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

132.8 Security

132.8.1 External Access

Repositories in general will get their metadata and artifacts from an external source, which makes them an attack vector for a malevolent Bundle that needs unauthorized external access. Since a Bundle using a Repository has no knowledge of what sources the Repository will access it will be necessary for the Repository to implement the external access in a doPrivileged block. Implementations must ensure that callers cannot influence/modify the metadata in such a way that the getContent() method could provide access to arbitrary Internet resources. This could for example happen if:

- The implementation relies on the osgi.content namespace to hold the URL
- The attributes Map from the osgi.content Capability is modifiable

If the malevolent Bundle could change the osgi.content attribute it could change it to arbitrary URLs. This example should make it clear that Repository implementations must be very careful.

132.8.2 Permissions

Implementations of this specification will need the following minimum permissions.

```
ServicePermission[...Repository, REGISTER ]
SocketPermission[ ... carefully restrict external access...]
```

Users of this specification will need the following minimum permissions.

```
ServicePermission[...Repository, GET ]
```

132.9 org.osgi.service.repository

Repository Service Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.repository; version="[1.1,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.repository; version="[1.1,1.2)"

132.9.1 Summary

- AndExpression - A RequirementExpression representing the and of a number of requirement expressions.
- ContentNamespace - Content Capability and Requirement Namespace.
- ExpressionCombiner - An ExpressionCombiner can be used to combine requirement expressions into a single complex requirement expression using the and, or and not operators.
- IdentityExpression - A RequirementExpression representing a requirement.
- NotExpression - A RequirementExpression representing the not (negation) of a requirement expression.
- OrExpression - A RequirementExpression representing the or of a number of requirement expressions.
- Repository - A repository service that contains resources.
- RepositoryContent - An accessor for the content of a resource.
- RequirementBuilder - A builder for requirements.
- RequirementExpression - The super interface for all requirement expressions.

132.9.2 public interface AndExpression extends RequirementExpression

A RequirementExpression representing the and of a number of requirement expressions.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.2.1 public List<RequirementExpression> getRequirementExpressions()

□ Return the requirement expressions that are combined by this AndExpression.

Returns An unmodifiable list of requirement expressions that are combined by this AndExpression. The list contains the requirement expressions in the order they were specified when this requirement expression was created.

132.9.3 public final class ContentNamespace extends Namespace

Content Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

132.9.3.1 public static final String CAPABILITY_MIME_ATTRIBUTE = "mime"

The capability attribute that defines the IANA MIME Type/Format for this content.

132.9.3.2 public static final String CAPABILITY_SIZE_ATTRIBUTE = "size"

The capability attribute that contains the size, in bytes, of the content. The value of this attribute must be of type Long.

132.9.3.3 public static final String CAPABILITY_URL_ATTRIBUTE = "url"

The capability attribute that contains the URL to the content.

132.9.3.4 public static final String CONTENT_NAMESPACE = "osgi.content"

Namespace name for content capabilities and requirements.

Also, the capability attribute used to specify the unique identifier of the content. This identifier is the SHA-256 hash of the content.

132.9.4 public interface ExpressionCombiner

An ExpressionCombiner can be used to combine requirement expressions into a single complex requirement expression using the and, or and not operators.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.4.1 public AndExpression and(RequirementExpression expr1, RequirementExpression expr2)

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

□ Combine two RequirementExpressions into a requirement expression using the and operator.

Returns An AndExpression representing an and of the specified requirement expressions.

132.9.4.2 public AndExpression and(RequirementExpression expr1, RequirementExpression expr2, RequirementExpression... moreExprs)

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

moreExprs Optional, additional requirement expressions to combine into the returned requirement expression.

□ Combine multiple RequirementExpressions into a requirement expression using the and operator.

Returns An AndExpression representing an and of the specified requirement expressions.

132.9.4.3 public IdentityExpression identity(Requirement req)

req The requirement to wrap in a requirement expression.

□ Wrap a Requirement in an IdentityExpression. This can be useful when working with a combination of Requirements and RequirementExpresions.

Returns An IdentityExpression representing the specified requirement.

132.9.4.4 public NotExpression not(RequirementExpression expr)

expr The requirement expression to negate.

□ Return the negation of a RequirementExpression.

Returns A NotExpression representing the not of the specified requirement expression.

132.9.4.5 public OrExpression or(RequirementExpression expr1, RequirementExpression expr2)

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

□ Combine two RequirementExpressions into a requirement expression using the or operator.

Returns An OrExpression representing an or of the specified requirement expressions.

132.9.4.6 public OrExpression or(RequirementExpression expr1, RequirementExpression expr2, RequirementExpression... moreExprs)

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

moreExprs Optional, additional requirement expressions to combine into the returned requirement expression.

□ Combine multiple RequirementExpressions into a requirement expression using the or operator.

Returns An OrExpression representing an or of the specified requirement expressions.

132.9.5 public interface IdentityExpression extends RequirementExpression

A RequirementExpression representing a requirement.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.5.1 public Requirement getRequirement()

□ Return the Requirement contained in this IdentityExpression.

Returns The requirement contained in this IdentityExpression.

132.9.6 public interface NotExpression extends RequirementExpression

A RequirementExpression representing the not (negation) of a requirement expression.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.6.1 public RequirementExpression getRequirementExpression()

□ Return the requirement expression that is negated by this NotExpression.

Returns The requirement expression that is negated by this NotExpression.

132.9.7 public interface OrExpression extends RequirementExpression

A RequirementExpression representing the or of a number of requirement expressions.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.7.1 public List<RequirementExpression> getRequirementExpressions()

□ Return the requirement expressions that are combined by this OrExpression.

Returns An unmodifiable list of requirement expressions that are combined by this OrExpression. The list contains the requirement expressions in the order they were specified when this requirement expression was created.

132.9.8 public interface Repository

A repository service that contains resources.

Repositories may be registered as services and may be used as by a resolve context during resolver operations.

Repositories registered as services may be filtered using standard service properties.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.8.1 public static final String URL = "repository.url"

Service property to provide URLs related to this repository.

The value of this property must be of type String, String[], or Collection<String>.

132.9.8.2 public Map<Requirement, Collection<Capability>> findProviders(Collection<? extends Requirement> requirements)

requirements The requirements for which matching capabilities should be returned. Must not be null.

□ Find the capabilities that match the specified requirements.

Returns A map of matching capabilities for the specified requirements. Each specified requirement must appear as a key in the map. If there are no matching capabilities for a specified requirement, then the value in the map for the specified requirement must be an empty collection. The returned map is the property of the caller and can be modified by the caller. The returned map may be lazily populated, so calling size() may result in a long running operation.

132.9.8.3 public Promise<Collection<Resource>> findProviders(RequirementExpression expression)

expression The RequirementExpression for which matching capabilities should be returned. Must not be null.

□ Find the resources that match the specified requirement expression.

Returns A promise to a collection of matching Resources. If there are no matching resources, an empty collection is returned. The returned collection is the property of the caller and can be modified by the caller. The returned collection may be lazily populated, so calling size() may result in a long running operation.

Since 1.1

132.9.8.4 public ExpressionCombiner getExpressionCombiner()

□ Return an expression combiner. An expression combiner can be used to combine multiple requirement expressions into more complex requirement expressions using and, or and not operators.

Returns An ExpressionCombiner.

Since 1.1

132.9.8.5 public RequirementBuilder newRequirementBuilder(String namespace)

namespace The namespace for the requirement to be created.

□ Return a new RequirementBuilder which provides a convenient way to create a requirement.

For example:

```
Requirement myReq = repository.newRequirementBuilder("org.foo.ns1").
  addDirective("filter", "(org.foo.ns1=val1)").
  addDirective("cardinality", "multiple").build();
```

Returns A new requirement builder for a requirement in the specified namespace.

Since 1.1

132.9.9 public interface RepositoryContent

An accessor for the content of a resource. All Resource objects which represent resources in a Repository must implement this interface. A user of the resource can then cast the Resource object to this type and then obtain an InputStream to the content of the resource.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.9.1 public InputStream getContent()

□ Returns a new input stream to the content of this resource. The content is represented on the resource through the osgi.content capability. If more than one such capability is associated with the resource, the first such capability is returned.

Returns A new input stream for associated content.

132.9.10 public interface RequirementBuilder

A builder for requirements.

Since 1.1

Provider Type Consumers of this API must not implement this type

132.9.10.1 public RequirementBuilder addAttribute(String name, Object value)

name The attribute name.

value The attribute value.

□ Add an attribute to the set of attributes.

Returns This requirement builder.

132.9.10.2 public RequirementBuilder addDirective(String name, String value)

name The directive name.

value The directive value.

□ Add a directive to the set of directives.

Returns This requirement builder.

132.9.10.3 public Requirement build()

□ Create a requirement based upon the values set in this requirement builder.

Returns A requirement created based upon the values set in this requirement builder.

132.9.10.4 public IdentityExpression buildExpression()

□ Create a requirement expression for a requirement based upon the values set in this requirement builder.

Returns A requirement expression created for a requirement based upon the values set in this requirement builder.

132.9.10.5 public RequirementBuilder setAttributes(Map<String, Object> attributes)

attributes The map of attributes.

□ Replace all attributes with the attributes in the specified map.

Returns This requirement builder.

132.9.10.6 public RequirementBuilder setDirectives(Map<String, String> directives)

directives The map of directives.

□ Replace all directives with the directives in the specified map.

Returns This requirement builder.

132.9.10.7 public RequirementBuilder setResource(Resource resource)

resource The resource.

□ Set the Resource.

A resource is optional. This method will replace any previously set resource.

Returns This requirement builder.

132.9.11 public interface RequirementExpression

The super interface for all requirement expressions. All requirement expressions must extend this interface.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.10 References

[1] *Framework Module Layer*
OSGi Core, Chapter 3 Module Layer

[2] *Framework Namespaces*
OSGi Core, Chapter 8, osgi.identity Namespace

[3] *Resource API Specification*
OSGi Core, Chapter 6 Resource API Specification

[4] *XML Schema Part 2: Data types Second Edition*
https://www.w3.org/TR/xmlschema-2/

[5] *XML Base (Second Edition), Resolving Relative URIs*
https://www.w3.org/TR/xmlbase/#resolution

[6] *Resolver Service Specification*
OSGi Core, Chapter 58 Resolver Service Specification

133 Service Loader Mediator Specification

Version 1.0

133.1 Introduction

Java SE 6 introduced the *Service Loader*, a simple service-provider loading facility, that attempted to unify the different ad-hoc mechanisms used by Java's many factories and builders. The design allows a JAR to advertise the name of one or more embedded classes that implement a given interface and consumers to obtain instances of these implementation classes through the Service Loader API.

Though the Service Loader is about extensibility, its own design is closed and therefore not extendable. It does not support a provider model that would allow different ways of finding interface implementations; its classes are final and its policy is fixed. Unfortunately, the Service Loader's fixed design uses a non-modular class loading policy; it defines its visibility scope with a class loader, which in general requires full visibility of the application's class path. The Service Loader can therefore in OSGi not find implementations from other bundles. Additionally, the Service Loader also does not enforce a life cycle; objects are handed out forever.

Since the Service Loader is the only standardized plugin mechanism in the JRE it is necessary that the mechanism is supported in OSGi with as few changes as possible from the consumer's authors. This specification therefore defines a *mediator* that ensures that the Service Loader is useful in an OSGi Framework, allowing programs that leverage the Service Loader to be used in OSGi frameworks almost as-is.

133.1.1 Essentials

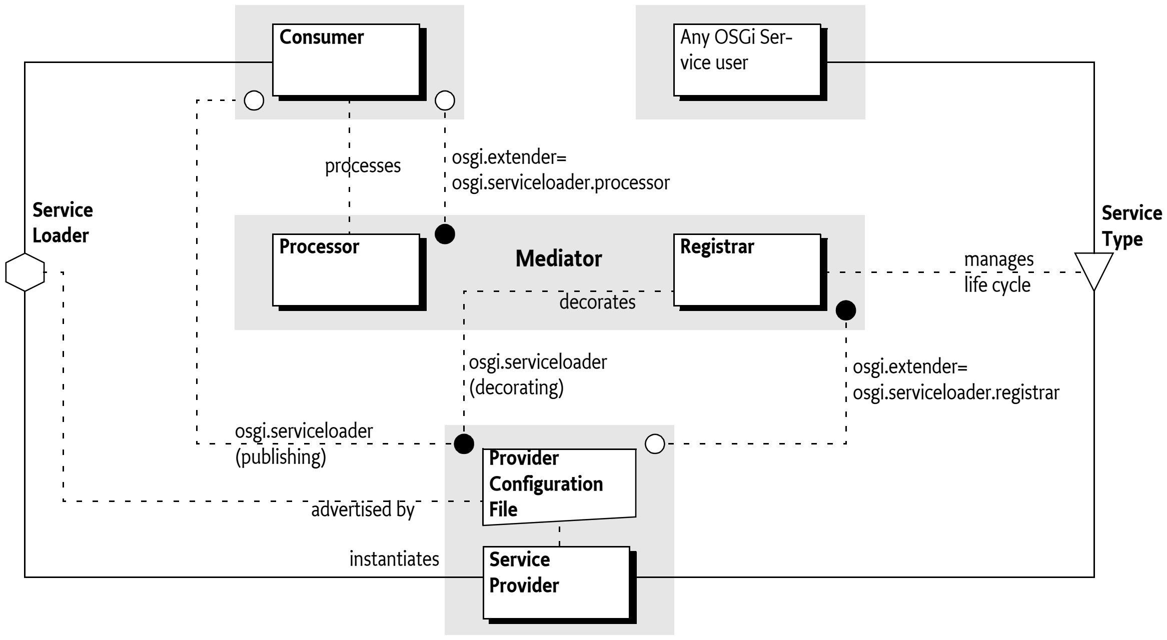
- *Compatibility* - Allow JARs that run in a classic Java SE environment that leverage the Service Loader to run in OSGi with only manifest modifications.
- *Services* - Register services for Service Provider bundles that opt-in.
- *Security* - Enforce service permissions for the Service Loader objects.
- *Life Cycle* - Manage the life cycle mismatch between OSGi bundles and the Service Loader's create only model.

133.1.2 Entities

- *Service Loader* - An API in Java SE that allows a Consumer to find an implementation of a Service Type from a Service Provider by searching a class loader for Service Providers.
- *Service Type* - The interface or class that the Service Provider must implement/extend.
- *Provider Configuration File* - A resource in the `META-INF/services` directory that has the fully qualified name of the Service Type and contains one ore more fully qualified names of Service Providers.
- *Service Provider* - An implementation class that implements or extends the Service Type.
- *Consumer* - A class that uses the Java SE Service Loader inside an OSGi framework.
- *Mediator* - An extender that mediates between Consumer bundles, the Service Loader API, and Service Provider bundles in an OSGi environment. It consists of a Processor and a Registrar.

- *Processor* - Modifies a bundle that uses the Service Loader API so that it works in an OSGi environment.
- *Registrar* - Registers services on behalf of a bundle that contains Service Providers.

Figure 133.1 *Entities*



133.1.3 Synopsis

This specification defines two different functions that are provided by a Mediator extender:

- Register OSGi services for each Service Provider.
- Allow Consumers that uses the Service Loader API to access Service Providers from other bundles that would normally not be visible from a bundle.

A Service Provider bundle can provide access to all its Service Providers through OSGi services by declaring a requirement on the `osgi.serviceloader.registrar` extender. This requirement activates a Mediator to inspect the `osgi.serviceloader` capabilities. If no `register` directive is used then all Service Providers for the given Service Type must be registered. Otherwise, each capability can select one Service Provider with the `register` directive. The fully qualified name selects a specific Service Provider, allowing different Service Providers to be registered with different service properties. The Mediator will then register an OSGi service factory for each selected capability. The `osgi.serviceloader` capability's attributes are used to decorate the OSGi service registration with service properties. The service factory returns a new instance for each service get.

Consumers are classes that use the Service Loader API to find Service Provider instances. Since the Service Loader API requires full visibility the Service API fails to work inside an OSGi bundle. A `osgi.serviceloader.processor` extender, which is the Mediator, processes bundles that require this capability by modifying calls to the Service Loader API to ensures that the Service Loader has visibility to published Service Providers.

A Consumer's bundle by default receives visibility to all published Service Providers. Service Providers are published when a bundle declares one or more `osgi.serviceloader` capabilities for a Service Type. If the Consumer has an `osgi.serviceloader` requirement for the given Service Type then the Mediator must only expose the bundles that are wired to those requirements and for each bundle provide all its Service Providers.

133.2 Java Service Loader API

Java is quite unique with its focus on separation of *specification* and *implementation*. Virtually all Java Specification Requests (JSR) provide a specification that can be implemented independently by different parties. Though this is one of the industry's best practices it raises a new problem: how to find the implementation available in a Java environment from only the *Service Type*. A Service Type is usually an interface but a base class can also be used.

Finding a Service Provider (the implementation class) from a Service Type is the so called *instance coupling* problem. The use of Service Types removed the type coupling between the Consumer of the contract and the *Service Provider* of the contract (the implementation) but to make things work there is a need of at least one place where the Service Provider is instantiated. The very large number of factories in Java reflects that this is a very common problem.

The general pattern for factories to find Service Providers was to search the class loaders for classes with constant names, varying the package names, often using System properties to extend the different areas to be sought. Though a general pattern based on class loading tricks emerged in the Java VM and application programs, all these patterns differed in details and places where they looked. This became harder and harder to maintain and often caused unexpected instances to be found.

The `java.util.ServiceLoader` class was therefore first introduced in Java SE 6 to provide a generic solution to this problem, see [1] *Java Service Loader API*. With this API Service Providers of a specification can now *advertise* their availability by creating a *Provider Configuration File* in their JAR in the `META-INF/services` directory. The name of this resource is the fully qualified name of the Service Type, the Service Provider provides when instantiated.

The Provider Configuration File contains a number of lines with comments or a class name that implements/extends the Service Type. For example:

```
org.example.Foo
```

A Service Provider must then advertise itself like:

```
META-INF/services/org.example.Foo:
    # Foo implementation
    org.acme.impl.FooImplementation
```

The Service Loader API finds all advertisers by constructing the name of the Provider Configuration File from the Service Type and then calling the `getResources` method on the provided class loader. This returns an enumeration of URLs to the advertisements. It then parses the contents of the resources; that will provide it with a list of Service Providers for the sought Service Type without duplicates. The API will return an iterator that will instantiate an object for the next available Service Provider.

To find the Configuration files for a given Service Type, the Service Loader uses a class loader. The Consumer can select the following class loaders:

- A given class loader as an argument in the call to the constructor
- The Thread Context Class Loader (TCCL)
- The system loader (when `null` is passed or no TCCL is set)

The class loader restricts the visibility of the Service Loader to only the resources to which the class loader has visibility. If the Service Loader has no access to the advertisement of a Service Provider then it cannot detect it and it will thus not be found.

The Service Provider is loaded from the given class loader, however, the `Class.forName` method is used, which stores it in the cache of the initiating class loader. This means that Service Providers are not garbage collected as long as there is a resolved bundle that used the Service Loader to get that Service Provider.

In the Service Loader API, the class does not have to originate from the same JAR file as the advertisement. In OSGi this is more restricted, the advertisement must come from the same bundle or must be explicitly imported.

For example, to load a Foo instance the following code could be used:

```
ServiceLoader<Foo> sl =
    ServiceLoader.load( Foo.class );
Iterator<Foo> it = sl.iterator();
if ( it.hasNext() ) {
    Foo foo = it.next();
    ...
}
```

Though the Service Loader API is about extensibility and contract based programming it is in itself not extendable nor replaceable. The ServiceLoader class is final, it comes from a sealed JAR, and is in a java package. It also does not provide an API to provide alternate means to find implementations for a Service Type.

133.3 Consumers

Consumers are classes that are not OSGi aware and directly use the *Service Loader API*. The Service Loader has a non-modular design and Consumers therefore run into many issues when running in an OSGi framework. Consumers should therefore in general be converted to use the OSGi service layer since this solves the visibility issues, life cycle impedance mismatch, and other problems. The Consumer part of this specification is therefore a last resort to use when existing code uses the Service Loader API and cannot be modified to leverage the OSGi service layer.

133.3.1 Processing

The Service Loader Mediator can *process* the Consumer by modifying calls to the Service Loader API. This specification does not detail how the Mediator ensures that the Consumer has visibility to other Service Providers. However, a Mediator could for example set an appropriate Thread Context Class Loader during the call to the Service Loader's constructor by weaving the Consumer's byte codes.

133.3.2 Opting In

Processing is an opt-in process, the Consumer bundle must declare that it is willing to be processed. The opt-in is handled by a requirement to the osgi.serviceloader.processor extender. This requirement must have a single cardinality (the default) since the Mediator uses the wiring to select the Consumer to process when multiple Mediators are present.

For example, the following requirement in a manifest enables a bundle to be processed:

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.processor)
                (version>=1.0)(!(version>=2.0)))"
```

If the extender osgi.serviceloader.processor requirement is satisfied then the wired Mediator must process the Consumer.

The Mediator must give visibility to all bundles with *published* Service Providers unless the Consumer restricts the visibility by having osgi.serviceloader requirements. Bundles publish a Service

Type, meaning all their Service Providers for that type, by having at least one osgi.serviceloader capability for that Service Type.

133.3.3 Restricting Visibility

A Consumer's bundle can restrict its visibility to certain bundles by declaring an osgi.serviceloader requirement for each Service Type it wants to use. Only bundles wired from those requirement provide their advertised Service Providers. If no such requirements are declared then all bundles with the published Service Type become available.

The cardinality can be used to select a single Service Provider's bundle or multiple bundles if it needs to see all Service Provider bundles. The requirement can be made optional if the Consumer's bundle can work also when no Service Provider bundle is available. See *osgi.serviceloader Namespace* on page 703 for more details.

For example, a requirement that restricts visibility to the org.example.Foo Service Providers could look like:

```
Require-Capability:
    osgi.serviceloader;
    filter:="(osgi.serviceloader=org.example.Foo)";
    cardinality:=multiple
```

In this example, any bundle that publishes the org.example.Foo Service Type will contribute its Service Providers.

Visibility can also be restricted to bundles that publish with capability's attributes. Any bundle that has at least one matching capability will then be able to contribute all its Service Providers. For example, the following example selects only bundles that have the classified property set:

```
osgi.serviceloader; filter:="(classified=*)"
```

With Service Registrations, see *Registering Services* on page 700, the capability can discriminate between multiple Service Providers in the same bundle. The Service Loader API does not have this feature: any wired requirement has visibility to all Service Providers in the wired bundle, regardless of the register directive.

133.3.4 Life Cycle Impedance Mismatch

A Consumer can only see Service Provider instances of bundles that are active during the time the next instance is created. That is, the Mediator must treat the life cycle of the Service Provider as if it was a service. However, the Service Loader implementations perform extensive class loader techniques and cache results. The exact life cycle of the Service Provider bundle with respect to the Consumer is therefore impossible to enforce.

The Service Loader API does not have a life cycle, objects are assumed to stay alive during the duration of the VM's process and due to the use of Class.forName in the Service Loader implementations. Therefore a Mediator should refresh a Consumer bundle when it is using a Service Provider and that Service Provider's bundle becomes stopped otherwise long running applications can run out of memory when bundles are regularly updated.

133.3.5 Consumer Example

A legacy JAR for which there is no more source code uses the Service Loader API to get access to com.example.Codec instances through the Service Loader API.

It is wrapped in a bundle that then has the following manifest:

```
Manifest-Version:       1.0
Bundle-ManifestVersion: 2
```

```
Bundle-SymbolicName:    com.example.impl
Bundle-Version:         23.98.1.v199101232310.02011
Import-Package:         com.example; version=3.45
Bundle-ClassPath:       legacy.jar
```

The manifest must then declare that the bundle must be processed, this is triggered by requiring the osgi.serviceloader.processor extender:

```
Require-Capability:
  osgi.extender;
    filter:="(&(osgi.extender=osgi.serviceloader.processor)
             (version>=1.0)(!(version>=2.0)))"
```

With this manifest, the Consumer bundle has full visibility to all Service Provider bundles that are published. The following lines can be added to restrict the visibility to codecs that have support for WAVE formats (although all Service Providers in that bundle will be visible to the consumer).

```
,
    osgi.serviceloader;
     filter:="(&(format=WAVE)(osgi.serviceloader=com.example.Codec))"
```

133.4 Service Provider Bundles

A *Service Provider bundle* is a bundle that contains one or more Service Providers that are usable by the Service Loader API. This section shows how Service Provider bundles should be constructed and what options they have.

133.4.1 Advertising

Service Providers are implementation classes that are *advertised* under a Service Type according to the rules in the Service Loader API. A Service Provider is advertised with a *Provider Configuration File* in a JAR. In an OSGi environment the Service Provider must reside in the same bundle as the advertisement or be imported. A single Provider Configuration File can contain multiple Service Providers. See *Java Service Loader API* on page 695.

133.4.2 Publishing the Service Providers

Service Providers can be used in two different scenarios:

- A Service Provider can be used by a processed Consumer as a Service Type, or
- It can be registered as a service.

A Service Type must be *published* to allow its use it in these scenarios. Publishing a Service Type consists of providing one or more osgi.serviceloader capabilities for an advertised Service Type, see *osgi.serviceloader Namespace* on page 703. These osgi.serviceloader capabilities must specify a fully qualified class name of the Service Type, there is no wildcarding allowed. Therefore, publishing a service implicitly makes all corresponding Service Providers available to Consumers.

If a bundle does not provide osgi.serviceloader capabilities then it does not publish any Service Providers and its Service Providers can therefore not be used by Consumers. They can then also not be registered as OSGi services, see *OSGi Services* on page 699. Tools can use the advertisement of the Service Provider in the JAR to automatically generate the osgi.serviceloader capabilities in the manifest.

For example, the following capability publishes all the Service Providers in its bundle that advertise the com.example.Codec interface:

```
Provide-Capability:
  osgi.serviceloader;
    osgi.serviceloader=com.example.Codec;
    uses:="com.example"
```

A Service Provider bundle must not require the osgi.serviceloader.processor extender unless it needs to be processed; publishing a Service Type is sufficient to allow Consumers to use the published Service Types.

133.4.3 OSGi Services

The Service Provider can have its osgi.serviceloader capabilities be registered as services that provide instances from the Service Providers. For this, the Service Provider bundle must require the osgi.serviceloader.registrar extender, which is the Mediator. For example:

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.registrar)
                 (version>=1.0)(!(version>=2.0)))"
```

The registrar must then inspect each osgi.serviceloader capability and register an associated OSGi Service for each Service Provider *selected* by that capability. A Service Provider is selected when:

- The capability has no register directive, or
- The register directive matches the fully qualified name of the Service Provider.

A register directive selects a Service Provider if it contains the fully qualified name of the Service Provider, that is, the implementation class. Selection only works for services, Consumer will always see all Service Providers regardless of the register directive due to limitations in the Service Loader API.

For example, the following manifest selects all Service Providers of the com.example.Foo Service Type since no register directive is present:

```
Provide-Capability:
    osgi.serviceloader;
        uses:="com.example";
        osgi.serviceloader=com.example.Foo
```

Selected Service Providers must be registered as defined in *Registering Services* on page 700, with the capability's attributes as *decorating* service properties. Private service properties (attributes that start with a full stop ('.' \u002E) and the defined capability attributes in the osgi.serviceloader namespace are not registered as service properties.

The following example would register the format service property but not the .hint service property for the com.acme.impl.WaveFoo Service Provider.

```
    osgi.serviceloader;
        osgi.serviceloader=com.example.Foo;
        uses:="com.example";
        format=WAVE;
        .hint=E5437Qy7;
        register:="com.acme.impl.WaveFoo"
```

The Mediator must only register OSGi services for selected Service Providers; the Service Provider bundle can therefore decide not to register certain Service Providers and register them with another mechanism, for example Declarative Services or in a bundle activator.

Since the Mediator must use the bundle context of the Service Provider to register the OSGi service the Service Provider bundle must have the proper Service Permission REGISTER for the Service Type.

133.4.4 Service Provider Example

A Foo Codecs JAR needs to be ported to OSGi, it provides a Service Provider for the org.example.Codec Service Type. In this example the JAR is given a new manifest:

```
Manifest-Version:       1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName:    com.example.foo.codecs
Import-Package:         com.example; version=3.45
```

To ensure that the bundle opts in to registering its services it must require the osgi.serviceloader.registrar extender.

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.registrar)
                 (version>=1.0)(!(version>=2.0)))"
```

To publish two Service Providers for the same type, two capabilities must be declared:

```
Provide-Capability:
    osgi.serviceloader;
        osgi.serviceloader="com.example.Codec";
        format:List<String>="WAVE,WMF";
      register:="com.acme.impl.FooWaveCodec";
      uses:="com.example,org.apache.common.codecs",
    osgi.serviceloader;
        osgi.serviceloader="com.example.Codec";
        format:List<String>=SINUS;
      register:="com.acme.impl.sinus.FooSinusCodec";
      uses:="com.example"
```

This example implicitly publishes the Service Type com.example.Codec multiple times with different attributes. Consumers that match any of these capabilities will however have visibility to all Service Providers since the Service Loader API cannot discriminate between different Service Providers from the same bundle.

133.5 Service Loader Mediator

A Mediator is the osgi.serviceloader.processor and osgi.serviceloader.registrar extender bundle that has the following responsibilities:

- It registers selected Service Providers as OSGi services.
- It processes any Consumers so that Service Loader API calls have proper visibility to published Service Provider bundles.

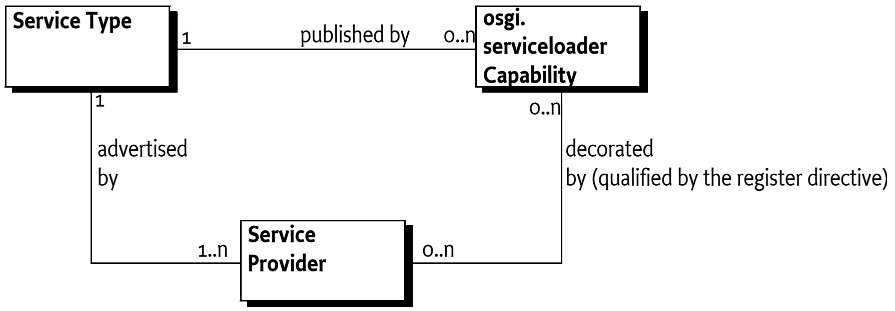
133.5.1 Registering Services

The Mediator must track bundles that are wired to its osgi.extender=osgi.serviceloader.registrar capability. These are called the *managed* bundles. For all managed bundles the Mediator must enumerate all osgi.serviceloader capabilities and register *selected* Service Providers as OSGi services. A Service Provider is selected by an osgi.serviceloader capability when:

- The advertised Service Type matches the corresponding `osgi.serviceloader` capability's Service Type, and
- The register directive is absent, or
  - The `register` directive contains the fully qualified name of the Service Provider.

An `osgi.serviceloader` capability that selects a Service Provider is said to *decorate* that Service Provider. A capability can decorate multiple Service Providers of the same Service Type and the same Service Provider can be decorated by different capabilities. Figure 133.2 depicts the resulting relations and their cardinalities since the relations are non-trivial.

Figure 133.2 *Cardinality Service Type*


Service Type
1
published by
0..n
osgi. serviceloader Capability
1
advertised by
0..n
decorated by (qualified by the register directive)
1..n
Service Provider
0..n

The OSGi service for each selected Service Provider must be registered under the advertised Service Type of the Service Provider, which must match the Service Type specified in the capability.

133.5.2 OSGi Service Factory

The Mediator must register an OSGi service factory with the bundle context of the Service Provider's bundle. The OSGi service factory must be implemented such that it creates a new instance for each bundle that gets the service. This behavior is similar, though not quite identical, to the `ServiceLoader.load()` method that gives each consumer a separate instance of the service. The difference is that different users inside a bundle will share the same instance.

Each service registration is controlled by a decorating `osgi.serviceloader` capability. The attributes on this capability must be registered with the OSGi service as service properties, except for:

- *Private* - Private properties, property names that start with a full stop ('.' `\u002E`) must not be registered.

The following service property must be registered, overriding any identical named properties in the decorating capability:

- `serviceloader.mediator` - (`Long`) The bundle id of the mediator.

The Mediator should not verify class space consistency since the OSGi framework already enforces this as long as the publishing capability specifies the `uses` directive.

Any services registered in the OSGi Service Registry must be unregistered when the Service Provider's bundle is stopped or the Mediator is stopped.

133.5.3 Service Loader and Modularity

The Service Loader API causes issues in a modular environment because it requires a class loader that has wide visibility. In a modular environment like OSGi the Consumer, the Service Type, and the Service Provider can, and should, all reside in different modules because they represent different concerns. Best practice requires that only the Service Type is shared between these actors. However, for the Service Loader to work as it was designed the Consumer must provide a class loader that has visibility of the Service Provider. The Service Provider is an implementation class, exporting such classes is the anathema of modularity. However, since the standard JRE provides application wide visibility this was never a major concern.

The simplest solution is to make the Service Loader aware of OSGi, its API clear is mappable to the OSGi service layer. However, the Service Loader is not extensible. The result is that using the Service Loader in OSGi fails in general because the Service Loader is unable to find the Service Providers. The issues are:

- The use of the Thread Context Class Loader (TCCL) is not defined in an OSGi environment. It should be set by the caller and this cannot be enforced. The multi threaded nature of OSGi makes it hard to predict what thread a Consumer will use, making it impossible to set an appropriate TCCL outside the Consumer.
- A bundle cannot import `META-INF/services` since the name is not a package name. Even if it could, the OSGi framework can only bind a single exporter to an importer for a given package. The Service Loader API requires access to all these pseudo-packages via the Class Loader's `getResources` method, the technique used to find Service Providers.
- Instantiating a Service Provider requires access to internal implementation classes, by exporting these classes, an implementing bundle would break its encapsulation.
- If a Service Provider was exported then importing this class in a Consumer bundle would couple it to a specific implementation package; this also violates the principle of loose coupling.
- The Service Loader API does assume an eternal life cycle, there is no way to signal that a Service Provider is no longer available. This is at odds with the dynamic bundle life cycle.

133.5.4 Processing Consumers

Consumers are not written for OSGi and require help to successfully use the Service Loader API. It is the Mediator's responsibility to ensure that bundles that are wired to published Service Types have access to these Service Provider's instances through the Service Loader API.

This specification does not define how this is done. There are a number of possibilities and it is up to the Mediator to provide the guarantee to the Consumer that it has been properly processed.

A Mediator must only process Consumer's bundles that are wired to the `osgi.extender` capability for the `osgi.serviceloader.processor` extender. Since Consumers must require this extender capability with the default cardinality of 1 there can at most be one extender wired to a Consumer.

133.5.5 Visibility

The Mediator must process the Consumer bundle in such a way that when the Consumer uses the Service Loader API it receives all the Service Providers of bundles that:

- Provide one or more `osgi.serviceloader` capabilities for the requested Service Type, and
- Are not type space incompatible with the requester for the given Service Type, and
- Either the Consumer has no `osgi.serviceloader` requirements or one of its requirements is wired to one of the `osgi.serviceloader` capabilities.

The Mediator must verify that the Consumer has Service Permission `GET` for the given Service Type since the Consumer uses the Service Type as a service. This specification therefore reuses the Service Permission for this purpose. The check must be done with the `ServicePermission(String,String)` constructor using the bundle's Access Control Context or the bundle's `hasPermission` method.

133.5.6 Life Cycle

There is a life cycle mismatch between the Service Loader API and the dynamic OSGi world. A Service Loader provides a Consumer with an object that could come from a bundle that is later stopped and/or refreshed. Such an object becomes *stale*. Mediators should attempt to refresh bundles that have access to these stale objects.

133.6 osgi.serviceloader Namespace

The osgi.serviceloader Namespace:

- Allows the Consumer's bundle to require the presence of a Service Provider for the required Service Type.
- Provides the service properties for the service registration.
- Indicates which Service Providers should be registered as an OSGi service.

The namespace is defined in the following table and ServiceLoaderNamespace, see *Common Namespaces Specification* on page 707 for the legend of this table.

Table 133.1 osgi.serviceloader namespace definition

Name	Kind	M/O	Type	Syntax	Description
osgi.serviceloader	CA	M	String	qname	The Service Type's fully qualified name.
*	CA	O	*	*	Additional matching attributes are permitted. These attributes will be registered as custom service properties unless they are private (start with a full stop).
register	CD	O	String	qname	Use this capability to register a different Service Factory under the Service Type for each selected Service Provider. A Service Provider is selected if the Service Type is the advertising Service Type and the Service Provider's fully qualified name matches the given name. If no register directive is present all advertised Service Providers must be registered. To register no Service Providers, because the capability must only be used to publish, provide an empty string.

133.7 Use of the osgi.extender Namespace

This section specifies the extender names for Mediators. They are used by both by Consumer and Service Provider bundles to ensure that a Mediator is present. Both names are defined for the general osgi.extender namespace in *osgi.extender Namespace* in *OSGi Core Release 8*.

The osgi.extender namespace requires the use of an *extender name*, the name of the Mediator extenders is:

```
osgi.serviceloader.processor
osgi.serviceloader.registrar
```

The version is for this specification is in both cases:

```
1.0.0
```

133.8 Security

133.8.1 Mediator

The Mediator will require significant permissions to perform its tasks. First, it will require access to the Bundle Context of the Service Provider bundle, which means it must have Admin Permission:

```
AdminPermission[<Service Provider Bundles>,CONTEXT|METADATA|CLASS]
```

Since it will have to register on behalf of the Service Provider bundle it must have complete liberty to register services:

```
ServicePermission[<Service Type>,REGISTER]
```

Depending on the way the Consumers are processed additional requirements may be necessary.

The Mediator connects two parties; it must ensure that neither party will receive additional permissions.

133.8.2 Consumers

Consumers must have:

```
ServicePermission[<Service Type>,GET]
PackagePermission[<Service Type's package>,IMPORT]
CapabilityPermission["osgi.extender", REQUIRE]
CapabilityPermission["osgi.serviceloader", REQUIRE]
```

The Mediator must ensure that the Consumer has the ServicePermission before it provides the instance. It must use the Bundle Context `hasPermission` method or the bundle's Access Control Context to verify this.

133.8.3 Service Providers

Service Providers must have:

```
ServicePermission[<Service Type>,REGISTER]
PackagePermission[<Service Type's package>,IMPORT]
CapabilityPermission["osgi.extender", REQUIRE]
CapabilityPermission["osgi.serviceloader", PROVIDE]
```

The Mediator must ensure that the Service Provider has the ServicePermission before it provides the instance. It must use the Bundle Context `hasPermission` method or the bundle's Access Control Context to verify this.

133.9 org.osgi.service.serviceloader

Service Loader Mediator Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.serviceloader; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.serviceloader; version="[1.0,1.1)"
```

133.9.1 Summary

- `ServiceLoaderNamespace` - Service Loader Capability and Requirement Namespace.

133.9.2 public final class ServiceLoaderNamespace extends Namespace

Service Loader Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

All unspecified capability attributes are of one of the following types:

- `String`
- `Version`
- `Long`
- `Double`
- `List<String>`
- `List<Version>`
- `List<Long>`
- `List<Double>`

and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type `String`, unless otherwise indicated.

All unspecified capability attributes, unless the attribute name starts with full stop ('.' \u002E), are also used as service properties when registering a Service Provider as a service.

Concurrency Immutable

133.9.2.1 public static final String CAPABILITY_REGISTER_DIRECTIVE = "register"

The capability directive used to specify the implementation class of the service. The value of this directive must be of type `String`.

If this directive is not specified, then all advertised Service Providers that match the service type name must be registered. If this directive is specified, then only Service Providers that match the service type name whose implementation class matches the value of this attribute must be registered. To not register a service for this capability use an empty string.

133.9.2.2 public static final String SERVICELOADER_NAMESPACE = "osgi.serviceloader"

Namespace name for service loader capabilities and requirements.

Also, the capability attribute used to specify the fully qualified name of the service type.

133.10 References

[1] *Java Service Loader API*
https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html

135 Common Namespaces Specification

Version 1.2

135.1 Introduction

A key aspect of the OSGi general dependency model based on requirements and capabilities is the concept of a *Namespace*. A Namespace defines the semantics of a Requirement-Capability pair. The generic model is defined in the [3] *Resources API Specification*. This section defines a number of Namespaces that are not part of the *OSGi Core Release 8* specification. Unless an attribute is specifically overridden, all Namespaces inherit the attributes and directives of the default Namespace as defined [4] *Framework Namespaces Specification*.

Each Namespace is defined with the following items:

- *Name* - the name of an attribute or directive
- *Kind* - Defines where the attribute or directive can be used
  - CA - Capability Attribute
  - CD - Capability Directive
  - RA - Requirement Attribute
  - RD - Requirement Directive
- *M/O* - Mandatory (M) or Optional (O)
- *Type* - The data type
- *Syntax* - Any syntax rules. The syntax refers in general to the syntaxes defined in [5] *General Syntax Definitions* and [6] *Common Headers*.

135.1.1 Versioning

In general, capabilities in a Namespace are versioned using Semantic Versioning. See [7] *Semantic Versioning*. Therefore, a capability will specify a single version and a requirement will specify a version range. See *osgi.extender Namespace* for an example.

For some Namespaces, capabilities are not versioned using Semantic Versioning. The versioning scheme used in those Namespaces will be described in the specification for the Namespace.

135.2 osgi.extender Namespace

An *Extender* is a bundle that uses the life cycle events from another bundle, the *extendee*, to extend that bundle's functionality when that bundle is active. It can use metadata (headers, or files inside the extendee) to control its functionality. Extendees therefore have a dependency on the Extender that can be modeled with the `osgi.extender` Namespace. The definition for this Namespace can be found in the following table and the `ExtenderNamespace` class.

Table 135.1 *osgi.extender Namespace*

Name	Kind	M/O	Type	Syntax	Description
osgi.extender	CA	M	String	symbolic-name	A symbolic name for the extender. These names are defined in their respective specifications and should in general use the specification top level package name. For example, org.acme.foo. The OSGi Working Group reserves names that start with "osgi.".
version	CA	M	Version	version	A version. This version must correspond to the specification of the extender.

Specifications for extenders (Blueprint, Declarative Services, etc.) should specify the values for these attributes. Extenders that provide such a capability should list the packages that they use in their specification in the uses directive of that capability to ensure class space consistency. For example a Declarative Services implementation could declare its capability with the following manifest header:

```
Provide-Capability: osgi.extender;
   osgi.extender="osgi.component";
   uses:="org.osgi.service.component";
   version:Version="1.3"
```

A bundle that depends on a Declarative Services implementation should require such an extender with the following manifest header:

```
Require-Capability: osgi.extender;
  filter:="(&(osgi.extender=osgi.component)(version>=1.3)(!(version>=2.0)))"
```

Extenders can extend an extendee bundle even if that bundle does not require the extender, unless the extender's specification explicitly forbids this. It is recommended that an extender should only extend a bundle if one of the following is true:

- The bundle's wiring has a required wire for at least one osgi.extender capability with the name of the extender and the first of these required wires is wired to the extender.
- The bundle's wiring has no required wire for an osgi.extender capability with the name of the extender.

Otherwise, the extender should not extend the bundle.

135.2.1 Extenders and Framework Hooks

The Framework provides a number of hooks that allow groups of bundles to be scoped. For example, the *Subsystem Service Specification.* An extender may want to extend the complete set of bundles installed in the Framework even when extendee bundles are hidden from the extender. The system bundle context provides a complete view of the bundles and services available in the Framework even if Framework hooks are used to scope groups of bundles. The system bundle context can be used by an extender to track all bundles installed in the Framework regardless of how Framework hooks are used to scope groups of bundles. This is useful in scenarios where several scoped groups contain bundles that require an extender. Instead of requiring an extender to be installed in each scoped group of bundles, a single extender that uses the system bundle context to track extendees can be installed to extend all scoped groups of bundles.

135.3 osgi.contract Namespace

Products or technologies often have a number of related APIs consisting of a large set of packages. Some IDEs have not optimized for OSGi and requires work for each imported package. In these development environments using modularized systems tends to require a significant amount of manual effort to manage the imported packages.

The osgi.contract Namespace addresses this IDE deficiency. It allows a developer to specify a single name and version for a contract that can then be expanded to a potentially large number of packages. For example, a developer can then specify a dependency on Java Enterprise Edition 6 contract that can be provided by an application server.

The osgi.contract Namespace provides such a name and binds it to a set of packages with the uses constraint. The bundle that declares this contract must then import or export each of the listed packages with the correct versioning. Such a bundle is called a *contract bundle*. The contract bundle must ensure that it is bound to the correct versions of the packages contained within the contract it is providing. If the contract bundle imports the packages which are specified as part of the contract then proper matching attributes must be used to make sure it is bound to the correct versions of the packages.

Additionally, the osgi.contract Namespace can be used in cases where API is defined by parties that do not use Semantic Versioning. In those cases, the version of the exported package can be unclear and so it is difficult to specify a meaningful version range for the package import. In such cases, importing the package *without* specifying a version range and specifying a requirement in the osgi.contract Namespace can provide a way to create portable bundles that use the API. OSGi has defined contract names for a number of such APIs. See [2] *Portable Java Contract Definitions* for more information.

An osgi.contract capability can then be used in the following ways:

- IDEs can use the information in the uses directive to make all those packages available on the build path. In this case the developer no longer has to specify each package separately.
- During run time the uses clause is used to enforce that all packages in the contract form a consistent class space.

The uses directive will make it impossible to get wired to packages that are not valid for the contract. Since the uses constrains enforce the consistency, it is in principle not necessary to version the imported packages on client bundles since only the correctly versioned packages can be used. Contracts are aggregates and therefore make clients depend on the whole and all their transitive dependencies, even if the client only uses a single package of the contract.

The recommended way of using contracts is to create a contract bundle that provides the osgi.contract capability and imports the packages with their required version range. For example:

```
Provide-Capability: osgi.contract;
    osgi.contract=JavaServlet;
    version:Version=2.5;
    uses:="javax.servlet,javax.servlet.http"
Export-Package:
    javax.servlet;      version="2.5",
    javax.servlet.http; version="2.5"
```

A contract may support multiple versions of a named contract. Such a contract must use a single capability for the contract name that specifies a list of all the versions that are supported. For example, the JavaServlet 3.1 contract capability would be specified with the following:

```
Provide-Capability: osgi.contract;
```

```
    osgi.contract=JavaServlet;
    version:List<Version>="2.5,3.0,3.1";
    uses:=
        "javax.servlet,
        javax.servlet.annotation,
        javax.servlet.descriptor,
        javax.servlet.http"
Export-Package:
    javax.servlet;            version="3.1",
    javax.servlet.annotation; version="3.1",
    javax.servlet.descriptor; version="3.1",
    javax.servlet.http;       version="3.1"
```

A client bundle that requires the Servlet 2.5 contract can then have the following manifest:

```
Require-Capability: osgi.contract;
    filter:="(&(osgi.contract=JavaServlet)(version=2.5))",
Import-Package:
    javax.servlet, javax.servlet.http
```

The client bundle will be constrained by the contract's uses constraints and automatically gets the correct packages. In this example, no semantic versioning is used for the contract because the Servlet Specifications do not use semantic versioning (version 3.0 is backward compatible with 2.X).

In this model it is even possible to use the normally not recommended DynamicImport-Package header with a wild card since also this header is constrained by the uses constraints. However, using a full wildcard can also dynamically import packages that are not part of the contract. To prevent these unwanted dynamic imports, the exporter could include an attribute on the exports. For example:

```
Require-Capability: osgi.contract;
    filter:="(&(osgi.contract=JavaServlet)(version=2.5))"
DynamicImport-Package:
    *;JavaServlet=contract
```

However, this model requires the exporter to specify an agreed attribute. The contract bundle does not require such coordination; it also allows the package exporters to reside in different and unrelated bundles.

The definition of the osgi.contract Namespace is in the following table and in the ContractNamespace class. See [2] *Portable Java Contract Definitions*.

Table 135.2 osgi.contract Namespace

Name	Kind	M/O	Type	Syntax	Description
osgi.contract	CA	M	String	symbolic-name	A symbolic name for the contract.
version	CA	O	Version+	version	A list of versions for the contract. A contract that supports multiple versions must use a single capability with a version attribute that lists all versions supported.
uses	CD	O	String	package-name (',' package-name)	For a contract, the standard uses clause is used to indicate which packages are part of the contract. The imports or exports of those packages link these packages to a particular version.

135.3.1 Versioning

As the osgi.contract Namespace follows the versioning of the associated contract, capabilities in this Namespace are *not* semantically versioned. The associated contracts are often versioned using marketing or other versioning schemes and therefore the version number cannot be used as an indication of backwards compatibility.

As a result, capabilities in the osgi.contract Namespace use a *discrete* versioning scheme. In such a versioning scheme, each version is treated as separate without any implied relation to another version. A capability lists *all* compatible versions. A requirement only selects a single version.

135.4 osgi.service Namespace

The Service Namespace is intended to be used for:

- Preventing a bundle from resolving if there is not at least one bundle that potentially can register a specific service.
- Providing a hint to the provisioning agent that the bundle requires a given service.
- Used as template for specifications like Blueprint and Declarative Services to express their provided and referenced services in the Repository model, see the *Repository Service Specification.*

A bundle providing this capability indicates that it can register such a service with at least the given custom attributes as service properties. At resolve time this is a promise since there is no guarantee that during runtime the bundle will actually register such a service; clients must handle this with the normal runtime dependency managers like Blueprint, Declarative Services, or others.

See the following table and the ServiceNamespace class for this Namespace definition.

Table 135.3 osgi.service Namespace

Name	Kind	M/O	Type	Syntax	Description
objectClass	CA	M	List <String>	qname (',' qname)*	The fully qualified name of the object class of the service.
*	CA	O	*	*	Custom attributes that will be provided as service properties if they do not conflict with the service properties rules and are not private service properties. Private properties start with a full stop ('.' \u002E).

135.4.1 Versioning

Capabilities in the osgi.service Namespace are *not* versioned. The package of a service's object class is generally versioned and the package can be associated with the capability via the uses directive.

135.5 osgi.implementation Namespace

The Implementation Namespace is intended to be used for:

- Preventing a bundle from resolving if there is not at least one bundle that provides an implementation of the specified specification or contract.
- Providing uses constraints to ensure that bundles which require an implementation of a specification or contract will be wired appropriately by the framework.
- Providing a hint to the provisioning agent that the bundle requires a given specification or contract implementation.
- Used as a general capability Namespace for specifications or contracts to express their provided function in the Repository model, see the *Repository Service Specification.*

A bundle providing this capability indicates that it implements a specification or contract with the specified name and version. For example, the *Asynchronous Service Specification* would provide the following capability:

```
Provide-Capability: osgi.implementation;
    osgi.implementation="osgi.async";
    version:Version="1.0";
    uses:="org.osgi.service.async"
```

See the following table and the ImplementationNamespace class for this Namespace definition.

Table 135.4 osgi.implementation Namespace

Name	Kind	M/O	Type	Syntax	Description
osgi.implementation	CA	M	String	symbolic-name	The symbolic name of the specification or contract. The OSGi Working Group reserves names that start with "osgi.".
version	CA	M	Version	version	The version of the implemented specification or contract.
*	CA	O	*	*	Custom attributes that can be used to further identify the implementation

135.6 osgi.unresolvable Namespace

The Unresolvable Namespace is intended to be used to mark a bundle as unresolvable:

- Preventing the bundle from resolving since it is intended for compilation use only and is not intended for runtime use.
- Providing a hint to the provisioning agent that the bundle must not be included in a provisioning solution.

For example, a bundle that must be unresolvable at runtime can include the following requirement:

```
Require-Capability: osgi.unresolvable;
  filter:="(&(must.not.resolve=*)(!(must.not.resolve=*)))"
```

The filter expression in the example above always evaluates to false.

See the UnresolvableNamespace class for this Namespace definition.

135.7 org.osgi.namespace.contract

Contract Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

135.7.1 Summary

- ContractNamespace - Contract Capability and Requirement Namespace.

135.7.2 public final class ContractNamespace extends Namespace

Contract Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

135.7.2.1 public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Versions of the specification of the contract. The value of this attribute must be of type Version, Version[], or List<Version>.

135.7.2.2 public static final String CONTRACT_NAMESPACE = "osgi.contract"

Namespace name for contract capabilities and requirements.

Also, the capability attribute used to specify the name of the contract.

135.8 org.osgi.namespace.extender

Extender Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

135.8.1 Summary

- ExtenderNamespace - Extender Capability and Requirement Namespace.

135.8.2 public final class ExtenderNamespace extends Namespace

Extender Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

135.8.2.1 public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Version of the specification of the extender. The value of this attribute must be of type Version.

135.8.2.2 public static final String EXTENDER_NAMESPACE = "osgi.extender"

Namespace name for extender capabilities and requirements.

Also, the capability attribute used to specify the name of the extender.

135.9 org.osgi.namespace.service

Service Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

135.9.1 Summary

- `ServiceNamespace` - Service Capability and Requirement Namespace.

135.9.2 public final class ServiceNamespace extends Namespace

Service Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

All unspecified capability attributes are of one of the following types:

- `String`
- `Version`
- `Long`
- `Double`
- `List<String>`
- `List<Version>`
- `List<Long>`
- `List<Double>`

and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type `String`, unless otherwise indicated.

Concurrency Immutable

135.9.2.1 public static final String CAPABILITY_OBJECTCLASS_ATTRIBUTE = "objectClass"

The capability attribute used to specify the types of the service. The value of this attribute must be of type `List<String>`.

A `ServiceNamespace` capability should express a uses constraint for all the packages mentioned in the value of this attribute.

135.9.2.2 public static final String SERVICE_NAMESPACE = "osgi.service"

Namespace name for service capabilities and requirements.

135.10 org.osgi.namespace.implementation

Implementation Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

135.10.1 Summary

- ImplementationNamespace - Implementation Capability and Requirement Namespace.

135.10.2 public final class ImplementationNamespace extends Namespace

Implementation Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

135.10.2.1 public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Version of the specification or contract being implemented. The value of this attribute must be of type Version.

135.10.2.2 public static final String IMPLEMENTATION_NAMESPACE = "osgi.implementation"

Namespace name for "implementation" capabilities and requirements. This is also the capability attribute used to specify the name of the specification or contract being implemented.

A ImplementationNamespace capability should express a uses constraint for the appropriate packages defined by the specification/contract the packages mentioned in the value of this attribute.

135.11 org.osgi.namespace.unresolvable

Unresolvable Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

135.11.1 Summary

- UnresolvableNamespace - Unresolvable Capability and Requirement Namespace.

135.11.2 public final class UnresolvableNamespace extends Namespace

Unresolvable Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

135.11.2.1 public static final String UNRESOLVABLE_FILTER = "(&(must.not.resolve=*)(!(must.not.resolve=*)))"

An unresolvable filter expression.

This can be used as the filter expression for an UnresolvableNamespace requirement.

```
@Requirement(namespace = UnresolvableNamespace.UNRESOLVABLE_NAMESPACE,
             filter = UnresolvableNamespace.UNRESOLVABLE_FILTER)
```

135.11.2.2 public static final String UNRESOLVABLE_NAMESPACE = "osgi.unresolvable"

Namespace name for "unresolvable" capabilities and requirements.

This is typically used as follows to prevent a bundle from being resolvable.

```
Require-Capability: osgi.unresolvable;
  filter:="(&(must.not.resolve=*)(!(must.not.resolve=*)))"
```

135.12 References

[1] *Specification References*
https://docs.osgi.org/reference/

[2] *Portable Java Contract Definitions*
https://docs.osgi.org/reference/portable-java-contracts.html

[3] *Resources API Specification*
OSGi Core, Chapter 6 Resource API Specification

[4] *Framework Namespaces Specification*
OSGi Core, Chapter 8 Framework Namespaces Specification

[5] *General Syntax Definitions*
OSGi Core, General Syntax Definitions

[6] *Common Headers*
OSGi Core, Chapter 3, Common Header Syntax

[7] *Semantic Versioning*
OSGi Core, Chapter 3, Semantic Versioning

137 REST Management Service Specification

Version 1.0

137.1 Introduction

Cloud computing is a continuing trend in the IT industry. Due to its service model which embraces dynamism as opposed to masking it, OSGi appears to be an ideal base for building scalable and dependable applications for the cloud where changes in the deployment, network topology, and service availability are the norm rather than the exception. One of the possible scenarios for OSGi to be successfully applied to cloud computing is using it in a Platform as a Service (PaaS) spirit. Users write their bundles and can deploy them to a provided OSGi instance running in the cloud. This, however, requires the platform provider to expose the OSGi management API to the end user and make them available through a network protocol. One of the popular approaches in cloud computing to remote communication is the use of RESTful web services.

Representational State Transfer (REST) is the architectural style of the world wide web. It can be described as a set of constraints that govern the interactions between the main components of the Internet. Recently, REST style interaction has gained popularity as a architecture for web services (RESTful web services), mainly to overcome the perceived complexity and verbosity of SOAP-based web services. This specification describes a REST interface for framework management, client-side Java and JavaScript APIs, and an extension mechanism through which other bundles can contribute their own RESTful management APIs and make them discoverable by clients.

137.1.1 Essentials

- *Client-Server* - A separation of concern between the entity responsible for the user-interaction (client) and the other entity (server) responsible for data storage. For instance, in the original world wide web the browser is the client rendering and presenting the content delivered by one or more web servers. As a result, web content becomes more portable and content providers more scalable.
- *Stateless* - State is entirely kept at the client side. Therefore, every request must contain all state required for the server to accomplish the transaction and deliver content. The main rationale behind this design constraint is to again improve the scalability since in a pure stateless design the server resources are not burdened with maintaining any client state. Another perceived advantage is that the failure models of stateless interactions is simpler and fault tolerance easier to achieve.
- *Cacheable* - Content marked as cacheable can be temporarily stored and used to immediately answer future equivalent requests and improve efficiency and reduce network utilization and access latencies. Due to the end-to-end principle, caches can be placed where necessary, e.g., at the client (forward-proxy), at the server side (backward-proxy), or somewhere in-between for example in a content delivery network. Content marked as non-cacheable must be freshly retrieved with every request even in the presence of caches.
- *Layered* - Layering introduces natural boundaries to coupling since every layer only accesses the services provided by the lower layer and provides services to the next higher layer.

- *Uniform Interface* - Generality of component interfaces provides a natural decoupling of implementation and interface. REST furthermore encourages the separation of identifiable resources (addressing) and their representation (content delivery).

137.1.2 Entities

- *Resource* - A resource is an abstract piece of information that can be addressed by a resource identifier. The mapping of a resource to a concrete set of entities can vary over time.
- *Representation* - A representation is a sequence of bytes plus associated meta-data that describe the state of a resource. The data format of a representation is called the media-type. Every concrete representation of a resource is just one of arbitrarily many possible representations. The selection of a concrete representation of a resource can be made according to the media types supported by both the client and the server.
- *REST Management Service* - The management service exposes a REST API for remotely managing an OSGi framework through the network in a lightweight and portable fashion.
- *Client* - The client is a machine using the management service by issuing REST requests through the network. It can do so either directly or indirectly, i.e., through client-side libraries using the REST calls internally.

137.1.3 Synopsis

The manageable entities of an OSGi framework are mapped to resources accessible through resource identifiers. These identifiers are relative to the (usually externally accessible) root URL of the management service. Clients can either discover this root URL or receive it through configuration. Subsequently, a client is able to introspect the state of the framework and perform management operations.

The internal state of a framework resource is expressed and transmitted as a representation. The format of the representation is subject to a mutual agreement between client and management service regarding media types commonly supported by both endpoints. This specification describes two representation formats: JSON and XML.

137.2 Interacting with the REST Management Service

The REST Management Service is not a traditional OSGi service and it does not appear in the service registry. Its purpose is to expose a management interface to clients which can perform operations on the framework through a network connection. Therefore, it is ideally suited for situations where the user of an OSGi framework does not have direct access to the machine it is running on, a typical situation in Infrastructure as a Service (IaaS) or Platform as a Service (PaaS). However, even in other domains having a lightweight and easily accessible management solution can be of benefit, e.g., for embedded devices. The advantage of REST is that it uses HTTP and therefore does usually not interfere with firewalls. Furthermore, the REST format is easily embeddable into client-side scripting technologies like JavaScript and can be consumed in web browsers.

Much of the value of the REST Management Service lies in client-side libraries which can use the REST protocol and interact with the OSGi framework through the Management Service. Therefore, this specification contains API for two clients, a Java Client API and a JavaScript Client API.

137.2.1 Resource Identifier Overview

The REST Management Service comprises of a set of resources that can be retrieved and in some cases also modified through REST requests. These resources need to be made available under well-defined paths so that clients can interact with them. As the initial entry point a client receives a URL to the REST Management Service. This can be done, e.g., as part of the creation of a cloud-based OSGi framework, and the precise mechanism would be proprietary to the cloud platform used. Relative to this URL the client can access the resources through the following resource identifiers:

```
framework
framework/state
framework/startlevel
framework/bundles

framework/bundles/representations

framework/bundle/{bundleid}
framework/bundle/{bundleid}/state
framework/bundle/{bundleid}/startlevel
framework/bundle/{bundleid}/header
framework/services

framework/services/representations

framework/service/{serviceid}
```

`framework/bundle/0/state` is an alias for `framework/state`

Extensions to the REST Management Service can be discovered by visiting the Extensions Resource at:

```
extensions
```

For more details on the extension mechanism see *Extending the REST Management Service* on page 731

137.2.2 Filtering Results

The `bundles`, `bundles/representations`, `services`, and `services/representations` resources allow the use of a query parameter which specifies a filter to restrict the result set. The filter expression follows the Core Specifications *Framework Filter Syntax*; see [1] *Framework Filter Syntax*.

Filters on services are matched against the service attributes. The query parameter is of the form:

`framework/services?filter=ldap-filter`

Filters on bundles are matched against the attributes of capabilities in the respective namespaces. Filters on bundles have the form:

`framework/bundles?namespace1=ldap-filter1&namespace2=ldap-filter2&...`

If multiple capabilities for a given namespace are present, then a filter succeeds when one of these capabilities matches. When multiple filter expressions across namespaces are given, these are combined with the *and* operator.

137.2.3 Content Type Matching

Resources can present themselves through different representation variants. An implementation of this specification must support at least the JSON representation and the XML representation of resources. Clients can support a subset of representations. Matching the clients capabilities to understand certain representation formats with the servers supported formats follows the typical HTTP pattern of content negotiation and requires the client to set corresponding HTTP Accept headers for supported formats in the form of their media types. This specification describes the format and media types for representations in JSON and XML format in *Representations* on page 725.

Implementations of the REST Management Service offering different variants of representations must return the best matching variant based on the HTTP accept header. In addition, they must respect the file extensions defined for the different media types as specified in the respective IETF RFC (e.g., ".xml" as specified in IETF RFC 3032 and ".json" as specified in IETF RFC 4627). If a file extension is appended to the resource, an implementation must return the variant mandated by the file extension provided that it supports this content type.

137.3 Resources

The framework and its state is mapped to a set of different resources. Each resource is accessible through a resource identifier, as summarized in *Resource Identifier Overview* on page 718.

137.3.1 Framework Startlevel Resource

framework/startlevel

The startlevel resource represents the active start level of the framework. It supports the GET and PUT requests.

137.3.1.1 GET

The GET request retrieves a *Framework Startlevel Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a startlevel representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.1.2 PUT

The PUT request sets the target framework startlevel. The body of the request needs to be a *Framework Startlevel Representation*. The request can return the following status codes:

- 204 (NO CONTENT): the request was received and valid. The framework will asynchronously start to adjust the framework startlevel until the target startlevel has been reached.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.
- 400 (BAD REQUEST): the REST management service received an IllegalArgumentException when trying to adjust the framework startlevel, e.g., because the requested startlevel was zero or negative.

137.3.2 Bundles Resource

framework/bundles

The bundles resource represents the list of all bundles installed on the managed framework. It supports the GET request and two syntactically different forms of POST requests which are used to install new bundles to the framework.

Results for this resource can be filtered as described in *Filtering Results* on page 719.

137.3.2.1 GET

The GET request retrieves a *Bundle List Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.2.2 POST with Location String

The POST request installs a new bundle to the managed framework and thereby logically appends it to the bundles resource. The new bundle to be installed is referenced by a location string which is passed as the body of the request. In order to disambiguate the request from the other form of POST, the content type must be set to text/plain. In practice, the location string is usually a URL. Since the framework will use the location retrieving the physical bundle, it needs to be accessible from the remotely managed framework and not necessarily from the managing client.

The management service implementation must check if the result of the install request matches the requested bundle since the OSGi framework will return an existing bundle object as the return value of an install call if there was already one with the same location string installed. One way of doing it is comparing the last modification timestamp. A detected collision is indicated to the requesting clients through an error code 409.

The body of the response is a *Bundle Representation* of the newly installed bundle. The following status codes can be returned:

- 200 (OK): the bundle has been successfully installed and the body of the response contains a *Bundle Representation*.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to install. The body of the message is a *Bundle Exception Representation* describing the reason why the installation did not succeed.
- 409 (CONFLICT): there is already a bundle installed with the same location string.

137.3.2.3 POST with Bundle

This variant of the POST request uploads the bundle as the body of the request. The media type of the request should be set to application/vnd.osgi.bundle which must be supported by all REST management services. Implementations are free to accept other media types for this request with the exception of text/plain. For instance, they can opt to additionally support application/zip or application/x-jar.

Clients should use the HTTP Content-Location field to set a bundle location. If no content location is given, REST management service implementations must generate a unique location string in order to avoid unintended collisions between unrelated bundles.

The body of the response is *Bundle Representation* of the newly installed bundle. The following status codes can be returned:

- 200 (OK): the bundle has been successfully installed and the body of the response contains the URI.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to install. The body of the message is a *Bundle Exception Representation* describing the reason why the installation did not succeed.
- 409 (CONFLICT): there is already a bundle installed with the same location string.

137.3.3 Bundles Representations Resource

framework/bundles/representations

137.3.3.1 GET of the Representations

The bundles resource returns a list of the URIs of all bundles installed on the framework. For clients interested in the details of multiple bundles there is also the possibility to retrieve the bundle representation of each installed bundle with a single request through the *bundles/representations* resource.

The body of the response is a *Bundle Representations List Representation*. The request can return the following status codes:

Results for this resource can be filtered as described in *Filtering Results* on page 719.

- 200 (OK): the request has been served successfully and the body of the response is a bundle list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.4 Bundle Resource

framework/bundle/{bundleid}

The bundle resource represents a single, distinct bundle in the system. Hence, it has to be qualified by a bundle id. The resource supports the GET, two variants of PUT, and the DELETE requests.

137.3.4.1 GET

The GET request retrieves a *Bundle Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.4.2 PUT with Location String

The PUT request updates the bundle with a new version, referenced by a location string which is passed as the body of the request. In order to disambiguate the request from the other form of PUT, the content type must be set to text/plain. The same rationale applies as for *POST with Location String* and *POST with Bundle* on page 721, if a location string is given it must point to a location reachable by the managed framework. If no location string is passed as the body of the request, the framework will perform an update based on the existing bundle's location string.

The body of the response is *Bundle Representation* of the updated bundle. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has issued the update.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to update. The body of the message is a *Bundle Exception Representation* describing the reason why the update did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.4.3 PUT with Bundle

The PUT request updates the bundle with a new version, uploaded as the body of the request. The media type of the request should be set to application/vnd.osgi.bundle which must be supported by all REST management services. Implementations are free to accept other media types for this request with the exception of text/plain. For instance, they can opt to additionally support application/zip or application/x-jar.

The body of the response is *Bundle Representation* of the updated bundle. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has issued the update.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to update. The body of the message is a *Bundle Exception Representation* describing the reason why the update did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.4.4 DELETE

The DELETE request uninstalls the bundle from the framework.

The body of the response is *Bundle Representation* of the uninstalled bundle, where the bundle state will be UNINSTALLED. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has uninstalled the bundle.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to uninstall. The body of the message is a *Bundle Exception Representation* describing the reason why the uninstallation did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.5 Bundle State Resource

`framework/bundle/{bundleid}/state`

The bundle state resource represents the internal state of an installed bundle qualified through its bundle id. It supports the GET and PUT requests.

137.3.5.1 GET

The GET request retrieves a *Bundle State Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle state representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.5.2 PUT

The PUT request sets the target state for the given bundle. This can, e.g., be state=32 for transitioning the bundle to started, or state=4 for stopping the bundle and transitioning it to resolved. The body of the request needs to be a *Bundle State Representation*. Not all state transitions are valid. The body of the response is the new *Bundle State Representation*. The request can return the following status codes:

- 200 (OK): the request was received and valid. The framework has performed a state change and the new bundle state is contained in the body.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to perform the state transition. The body of the message is a *Bundle Exception Representation* describing the reason why the operation did not succeed.
- 402 (PRECONDITION FAILED): the requested target state is not reachable from the current bundle state or is not a target state. An example such state is the STOPPING state.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.

137.3.6 Bundle Header Resource

`framework/bundle/{bundleid}/header`

The bundle header resource represents manifest header of a bundle which is qualified by its bundle id. It can only be read through a GET request.

137.3.6.1 GET

The GET request retrieves a *Bundle Header Representation* from the REST management service. The raw header value is used unless an Accept-Language header is set on the HTTP request. If multiple accepted languages are set only the first is used to localize the header. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle header representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.7 Bundle Startlevel Resource

framework/bundle/{bundleid}/startlevel

The bundle startlevel resource represents the start level of the bundle qualified by its bundle id. It supports the GET and PUT requests.

137.3.7.1 GET

The GET request retrieves a *Bundle Startlevel Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle startlevel representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.7.2 PUT

The PUT request sets the target bundle startlevel. The body of the request needs to be a *Bundle Startlevel Representation*, however only the startLevel property is used. The request can return the following status codes:

- 200 (OK): the request was received and valid. The REST management service has changed the bundle startlevel according to the target value. The body of the response is the new bundle startlevel representation.
- 400 (BAD REQUEST): either the target startlevel state involved invalid values, e.g., a startlevel smaller or equal to zero and the REST management service got an IllegalArgumentException, or the REST management service received a BundleException when trying to perform the startlevel change. In the latter case, the body of the message is a *Bundle Exception Representation* describing the reason why the operation did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.

137.3.8 Services Resource

framework/services

The services resource represents the set of all services available on the framework, optionally constrained by a filter expression. It is read-only and therefore only supports the GET request.

Results for this resource can be filtered as described in *Filtering Results* on page 719.

137.3.8.1 GET

The GET request retrieves a *Service List Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a service list representation.
- 400 (BAD REQUEST): the provided filter expression was not valid.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.9 Services Representations Resource

```
framework/services/representations
```

137.3.9.1 GET of the Representations

The services resource returns a list of the URIs of all services registered on the framework. For clients interested in the details of multiple services there is also the possibility to retrieve the service representation of each available service with a single request through the *services/representations* resource. The body of the response is a *Service Representations List Representation* from the REST management service. The request can return the following status codes:

Results for this resource can be filtered as described in *Filtering Results* on page 719.

- 200 (OK): the request has been served successfully and the body of the response is a service list representation.
- 400 (BAD REQUEST): the provided filter expression was not valid.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.10 Service Resource

```
framework/service/{serviceid}
```

The service resource represents a single, distinct service in the framework. Hence, it has to be qualified by a service id. Services can only be read through the REST Management Service and therefore only support the GET request.

137.3.10.1 GET

The GET request retrieves a *Service Representation* . The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a service representation.
- 404 (NOT FOUND): there is not service with the given service id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.4 Representations

137.4.1 Bundle Representation

137.4.1.1 JSON

```
Content-Type: application/org.osgi.bundle+json
```

```
{
   "id":0,
   "lastModified":1314999275542,
   "state":32,
   "symbolicName":"org.eclipse.osgi",
   "version":"3.7.0.v20110613"
}
```

137.4.1.2 XML

Content-Type: application/org.osgi.bundle+xml

```
<bundle>
   <id>0</id>
   <lastModified>1314999275542</lastModified>
   <state>32</state>
   <symbolicName>org.eclipse.osgi</symbolicName>
   <version>3.7.0.v20110613</version>
</bundle>
```

137.4.2 Bundles Representations

137.4.2.1 Bundle List Representation

137.4.2.1.1 JSON

Content-Type: application/org.osgi.bundles+json

```
[bundleURI, bundleURI, ..., bundleURI]
```

137.4.2.1.2 XML

Content-Type: application/org.osgi.bundles+xml

```
<bundles>
   <uri>bundleURI</uri>
   <uri>bundleURI</uri>
   ...
   <uri>bundleURI</uri>
</bundles>
```

137.4.2.2 Bundle Representations List Representation

137.4.2.2.1 JSON

Content-Type: application/org.osgi.bundles.representations+json

```
[BUNDLE REPRESENTATION, BUNDLE REPRESENTATION, ..., BUNDLE REPRESENTATION]
```

137.4.2.2.2 XML

Content-Type: application/org.osgi.bundles.representations+xml

```
<bundles>
   BUNDLE REPRESENTATION
   BUNDLE REPRESENTATION
   ...
   BUNDLE REPRESENTATION
</bundles>
```

137.4.3 Bundle State Representation

137.4.3.1 JSON

Content-Type: application/org.osgi.bundlestate+json

```
{
   "state":32
   "options":1
}
```

The options are used in start or stop calls. Valid options include, e.g., Bundle.START_TRANSIENT and Bundle.START_ACTIVATION_POLICY.

137.4.3.2 XML

Content-Type: application/org.osgi.bundlestate+xml

```
<bundleState>
   <state>32</state>
   <options>1</options>
</bundleState>
```

137.4.4 Bundle Header Representation

137.4.4.1 JSON

Content-Type: application/org.osgi.bundleheader+json

```
{
   key:value,
   key:value,
   ...
   key:value
}
```

137.4.4.2 XML

Content-Type: application/org.osgi.bundleheader+xml

```
<bundleHeader>
   <entry key="key" value="value"/>
   <entry key="key" value="value"/>
   ...
   <entry key="key" value="value"/>
<bundleHeader>
```

137.4.5 Framework Startlevel Representation

137.4.5.1 JSON

Content-Type: application/org.osgi.frameworkstartlevel+json

```
{
    "startLevel":6,
    "initialBundleStartLevel":4
}
```

137.4.5.2 XML

Content-Type: application/org.osgi.frameworkstartlevel+xml

```
<frameworkStartLevel>
    <startLevel>6</startLevel>
    <initialBundleStartLevel>4</initialBundleStartLevel>
</frameworkStartLevel>
```

137.4.6 Bundle Startlevel Representation

137.4.6.1 JSON

Content-Type: application/org.osgi.bundlestartlevel+json

```
{
   "startLevel":6

   "activationPolicyUsed":true
   "persistentlyStarted":false
}
```

137.4.6.2 XML

Content-Type: application/org.osgi.bundlestartlevel+xml

```
<bundleStartLevel>
    <startLevel>6</startLevel>

    <activationPolicyUsed>true</actiovationPolicyUsed>
    <persistentlyStarted>false</persistentlyStarted>
</bundleStartLevel>
```

137.4.7 Service Representation

137.4.7.1 JSON

Content-Type: application/org.osgi.service+json

```
{
   "id":10,
   "properties":
   {
      "prop1":"val1",
      "prop2":2.82,
      ...
      "prop3":true
   },
   "bundle":bundleURI,
   "usingBundles":[bundleURI, bundleURI, ... bundleURI]
}
```

Note: service properties are converted to JSON-supported data types where possible: "string", number or boolean (true|false). If there is no conversion to JSON data types is possible the toString() result is used as a string value.

137.4.7.2 XML

Content-Type: application/org.osgi.service+xml

```
<service>
   <id>10</id>
   <properties>
      <property name="prop1" value="val1"/>
      <property name="prop2" type="Float" value="2.82"/>
      ...
      <property name="prop3" type="Boolean" value="true"/>
   </properties>
   <bundle>bundleURI</bundle>
   <usingBundles>
      <bundle>bundleURI</bundle>
      <bundle>bundleURI</bundle>
      ...
      <bundle>bundleURI</bundle>
   </usingBundles>
</service>
```

Note: service properties are represented using the same method as used for the property XML element in the Declarative Services specification, see *Property and Properties Elements* on page 268. Service properties that cannot be represented using the supported data types, will be represented as String values obtained via the `toString()` method.

137.4.8 Services Representations

137.4.8.1 Service List Representation

137.4.8.1.1 JSON

Content-Type: application/org.osgi.services+json

```
[serviceURI, serviceURI, ..., serviceURI]
```

137.4.8.1.2 XML

Content-Type: application/org.osgi.services+xml

```
<services>
   <uri>serviceURI</uri>
   <uri>serviceURI</uri>
   ...
   <uri>serviceURI</uri>
</services>
```

137.4.8.2 Service Representations List Representation

137.4.8.2.1 JSON

Content-Type: org.osgi.services.representations+json

```
[SERVICE REPRESENTATION, SERVICE REPRESENTATION, ..., SERVICE REPRESENTATION]
```

137.4.8.2.2 XML

Content-Type: application/org.osgi.services.representations+xml

```
<services>
   SERVICE REPRESENTATION
   SERVICE REPRESENTATION
   ...
   SERVICE REPRESENTATION
</services>
```

137.4.9 Bundle Exception Representation

137.4.9.1 JSON

Content-Type: application/org.osgi.bundleexception+json

```
{
   "typecode": 5,
   "message": "BundleException: Bundle activation error"
}
```

137.4.9.2 XML

Content-Type: application/org.osgi.bundleexception+xml

```
<bundleexception>
   <typecode>5</typecode>
   <message>BundleException: Bundle activation error</message>
</bundleexception>
```

137.5 Clients

The REST service can be used by a variety of clients directly. In addition this specification describes Client APIs built over this REST protocol to facilitate use from Java and JavaScript clients.

137.5.1 Java Client

The Java Client provides a Java API over the REST API providing a convenient and portable way to use this API from a Java application.

To use the Java Client, obtain the RestClientFactory service. Create a client by providing the root URL of the REST service, for example:

```
RestClientFactory restClientFactory = ... // from Service Registry
RestClient restClient = restClientFactory.createRestClient(
    new URI("http://localhost:8080/restendpoint"));

// Now we can start interacting
Collection<String> bundles = restClient.getBundlePaths();
BundleDTO newBundle = restClient.installBundle(bundleLocation, bundleStream);
restClient.startBundle(newBundle.id);
```

The more details on the Java Client can be found in the org.osgi.service.rest.client API documentation section.

137.5.2 JavaScript Client

This specification also describes a JavaScript client to the REST Management service. This client makes it easy to manage an OSGi framework from any JavaScript environment, including Web Browsers.

The JavaScript client follows the *promises* programming style; the request is made asynchronously and a success() or failure() callback is made when the response arrives.

To use the JavaScript client create an instance of OSGiRestClient providing the root URL of the REST service.

```
var client = new OSGiRestClient('http://localhost:8080/restendpoint');
client.installBundle({
  success : function(res) {
    // Start the bundle once the install has finished
    client.startBundle(res.id);
  },
  failure : function(httpCode, res) {
    // handle failure
  }
});
```

More details on the JavaScript Client can be found in the JavaScript Client API API documentation section.

137.6 Extending the REST Management Service

This specification describes a REST-based management interface for Core Framework functionality. Other services in the framework might also benefit from management access through REST. This can involve services specified by the OSGi Working Group as part of the Core or Compendium Specifications but also application-specific functionality provided by the developer. It is desirable to expose such management services as extensions of the REST Management Service.

This REST service can be implemented by using various technologies such as Java Servlets, Restlet, JAX-RS, and others. Therefore, it might not always be possible to integrate extensions at the implementation level because they might use other underlying technologies to implement their REST interface. Defining a format for delegating requests between the REST Management Service and extensions would furthermore necessarily expose implementation details and is therefore not feasible either. As a consequence, this specification only describes how to logically integrate extensions with the REST Management Service. Implementations of this specification might offer mechanisms for tighter integration for the case that extensions are developed using the same underlying technology.

The main purpose of the extension mechanism is to advertise extensions to the core REST implementation, which makes them discoverable for clients. This mechanism can be used to check if a REST interface exists for a specific service. This is done through the *Extensions Resource* which contains a description and a path for every extension currently available. Implementations that want to contribute their extensions to the REST Management Service can do so by registering the RestApiExtension service using the [4] *Whiteboard Pattern*. The extension interface is only a marker and the relevant information is exposed through the NAME, URI_PATH and optionally SERVICE properties. Note that it is the responsibility of the extension to ensure that the endpoint announced via the RestApiExtension service is actually present. The Whiteboard service does not realize the extension endpoint; it purely announces it to the main REST implementation for inclusion in the Extensions Resource.

In order to be discoverable REST interface extensions to OSGi Core or Compendium services must use their canonical package name as advertised name. E.g., the name of the REST interface for the User Admin must be org.osgi.service.useradmin. This way, a client is able to check if there is a given extension available on a host. User-defined extensions should use the package name of the service they provide management capabilities for.

137.6.1 Extensions Resource

extensions

The extensions resource enumerates all extensions currently registered through the Whiteboard Pattern. It is read-only and therefore only supports the GET request.

137.6.1.1 GET

The GET request retrieves a *Extensions Representation* . The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a extension list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.6.2 Extensions Representation

137.6.2.1 JSON

Content-Type: application/org.osgi.extensions+json

```
[ { "name" : "org.osgi.service.event", "path" : "contributions/eventadmin",
      "service" : 12 }, ... ]
```

137.6.2.2 XML

Content-Type: application/org.osgi.extensions+xml

```
<extensions>
   <extension>
      <name>org.osgi.service.event</name>
      <path>contributions/eventadmin</path>
      <service>12</service>
   </extension>
</extensions>
```

137.7 XML Schema

The namespace for XML representations is:

`http://www.osgi.org/xmlns/rest/v1.0.0`

The recommended prefix for this namespace is rest.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:rest="http://www.osgi.org/xmlns/rest/v1.0.0"
    targetNamespace="http://www.osgi.org/xmlns/rest/v1.0.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.0.0">

    <annotation>
        <documentation xml:lang="en">
            This is the XML Schema for
            XML representations used by
            the REST Management Service
            Specification.
        </documentation>
    </annotation>

    <element name="bundle" type="rest:Tbundle">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundle+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="Tbundle">
        <all>
            <element name="id" type="long" />
            <element name="lastModified" type="long" />
            <element name="state" type="integer" />
            <element name="symbolicName" type="string" />
            <element name="version" type="string" />
        </all>
    </complexType>

    <element name="bundles" type="rest:Tbundles">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundles+xml and
                application/org.osgi.bundles.representations+xml content
                types.
            </documentation>
        </annotation>
    </element>
    <complexType name="Tbundles">
        <choice>
            <element name="uri" type="string" minOccurs="0"
                maxOccurs="unbounded">
                <annotation>
                    <documentation xml:lang="en">
                        Representation for the
                        application/org.osgi.bundles+xml content type.
                    </documentation>
                </annotation>
            </element>
            <element name="bundle" type="rest:Tbundle"
                minOccurs="0" maxOccurs="unbounded">
                <annotation>
                    <documentation xml:lang="en">
                        Representation for the
                        application/org.osgi.bundles.representations+xml
                        content type.
                    </documentation>
                </annotation>
            </element>
        </choice>
    </complexType>

    <element name="bundleState" type="rest:TbundleState">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundlestate+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="TbundleState">
        <all>
            <element name="state" type="integer" />
            <element name="options" type="integer" />
        </all>
    </complexType>

    <element name="bundleHeader" type="rest:TbundleHeader">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundleheader+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="TbundleHeader">
        <sequence>
            <element name="entry" minOccurs="0" maxOccurs="unbounded">
                <complexType>
                    <attribute name="key" type="string" use="required" />
                    <attribute name="value" type="string" use="required" />
                </complexType>
            </element>
        </sequence>
    </complexType>

    <element name="frameworkStartLevel" type="rest:TframeworkStartLevel">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.frameworkstartlevel+xml content
                type.
            </documentation>
        </annotation>
    </element>
    <complexType name="TframeworkStartLevel">
        <all>
```

```
            <element name="startLevel" type="integer" />
            <element name="initialBundleStartLevel" type="integer" />
        </all>
    </complexType>

    <element name="bundleStartLevel" type="rest:TbundleStartLevel">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundlestartlevel+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="TbundleStartLevel">
        <all>
            <element name="startLevel" type="integer" />
            <element name="activationPolicyUsed" type="boolean" />
            <element name="persistentlyStarted" type="boolean" />
        </all>
    </complexType>

    <element name="service" type="rest:Tservice">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.service+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="Tservice">
        <all>
            <element name="id" type="long" />
            <element name="properties">
                <complexType>
                    <sequence>
                        <element name="property" minOccurs="0"
                            maxOccurs="unbounded">
                            <complexType>
                                <simpleContent>
                                    <extension base="string">
                                        <attribute name="name"
                                            type="string" use="required" />
                                        <attribute name="value"
                                            type="string" use="optional" />
                                        <attribute name="type"
                                            default="String" use="optional">
                                            <simpleType>
                                                <restriction
                                                    base="string">
                                                    <enumeration
                                                        value="String" />
                                                    <enumeration
                                                        value="Long" />
                                                    <enumeration
                                                        value="Double" />
                                                    <enumeration
                                                        value="Float" />
                                                    <enumeration
                                                        value="Integer" />
                                                    <enumeration
                                                        value="Byte" />
                                                    <enumeration
                                                        value="Character" />
                                                    <enumeration
                                                        value="Boolean" />
                                                    <enumeration
                                                        value="Short" />
                                                </restriction>
                                            </simpleType>
                                        </attribute>
                                    </extension>
                                </simpleContent>
                            </complexType>
                        </element>
                    </sequence>
                </complexType>
            </element>
            <element name="bundle" type="string" />
            <element name="usingBundles">
                <complexType>
                    <sequence>
                        <element name="bundle" type="string"
                            minOccurs="0" maxOccurs="unbounded" />
                    </sequence>
                </complexType>
            </element>
        </all>
    </complexType>

    <element name="services" type="rest:Tservices">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.services+xml and
                application/org.osgi.services.representations+xml
                content types.
            </documentation>
        </annotation>
    </element>
    <complexType name="Tservices">
        <choice>
            <element name="uri" type="string" minOccurs="0"
                maxOccurs="unbounded">
                <annotation>
                    <documentation xml:lang="en">
                        Representation for the
                        application/org.osgi.services+xml content type.
                    </documentation>
                </annotation>
            </element>
            <element name="service" type="rest:Tservice"
                minOccurs="0" maxOccurs="unbounded">
                <annotation>
                    <documentation xml:lang="en">
                        Representation for the
                        application/org.osgi.services.representations+xml
                        content type.
                    </documentation>
                </annotation>
            </element>
        </choice>
    </complexType>

    <element name="bundleexception" type="rest:Tbundleexception">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.bundleexception+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="Tbundleexception">
        <all>
            <element name="typecode" type="integer" />
            <element name="message" type="string" />
        </all>
    </complexType>

    <element name="extensions" type="rest:Textensions">
        <annotation>
            <documentation xml:lang="en">
                Representation for the
                application/org.osgi.extensions+xml content type.
            </documentation>
        </annotation>
    </element>
    <complexType name="Textensions">
        <sequence>
            <element name="extension" minOccurs="0" maxOccurs="unbounded">
                <complexType>
```

```
                        <all>
                            <element name="name" type="string" />
                            <element name="path" type="string" />
                            <element name="service" type="long" minOccurs="0" />
                        </all>
                    </complexType>
                </element>
            </sequence>
        </complexType>
</schema>
```

The schema is also available in digital form from [3] *OSGi XML Schemas*.

137.8 Capabilities

137.8.1 osgi.implementation Capability

An implementation of this specification must provide the osgi.implementation capability with name osgi.rest. This capability can be used by provisioning tools and during resolution to ensure that a REST Management implementation is present to handle REST requests defined in this specification. The capability must also declare a uses constraint on the org.osgi.service.rest package:

```
Provide-Capability: osgi.implementation;
  osgi.implementation="osgi.rest";
  uses:="org.osgi.service.rest";
  version:Version="1.0"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

137.8.2 osgi.service Capability

A bundle providing the RestClientFactory service as described by this specification must inform tools about this service by providing the osgi.service capability representing this service. This capability must also declare a uses constraint for the org.osgi.service.rest.client package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.rest.client.RestClientFactory";
  uses:="org.osgi.service.rest.client"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

137.9 Security

Like any externally visible management interface, the REST interface exposes privileged operations and hence requires access control. Since REST builds upon the HTTP(s) protocol, authentication mechanisms and encryption can be applied the same way as usually done for web servers: they can be layered below the REST protocol. E.g., confidentiality of the transmitted commands can be ensured by using HTTPS as the underlying transport. Authentication can be added by requiring, e.g., basic authentication prior to accepting a REST command. The REST interface should only be implemented by a trusted bundle. Implementations of this specification require all Admin Permissions and all Service Permissions.

137.10 org.osgi.service.rest

Rest Service Package Version 1.0.

137.10.1 Summary

- `RestApiExtension` - Marker interface for registering extensions to the Rest API service.

137.10.2 public interface RestApiExtension

Marker interface for registering extensions to the Rest API service.

The REST service provides a RESTful interface to clients that need to manage an OSGi framework through a network connection. Other components running on the same framework can contribute their own specific REST interface and make it available and discoverable by registering this marker service using the Whiteboard pattern.

Integration of third-party REST interfaces with the framework REST service on the implementation level might not always be possible since it requires knowledge about the underlying implementation and an extension mechanism on that level. Specific technologies such as servlets might support this but the REST service could as well be implemented without the use of a supporting abstraction layer and not offer extensibility.

Using this marker service, the REST service includes the advertised service in the Extensions Resource, allowing clients to discover it and use the extension's functionality.

137.10.2.1 public static final String NAME = "org.osgi.rest.name"

This service property describes the package name of the technology manageable by this REST API extension. Services specified in OSGi specifications must use their canonical package name as the name. Third-party technologies should also use their package names. The type of this property is `java.lang.String` and the property is mandatory.

137.10.2.2 public static final String SERVICE = "org.osgi.rest.service"

This service property refers to the id of the service the REST API extension provides management capabilities for. This can be useful if more than one service of a given type is present in the framework. For example if more than one Http Service is available this property is used to associate a REST extension managing the Http Service with a specific service instance. The type of the property is `java.lang.Long` and the property is optional; if the REST extension is not directly associated with a service in the service registry, the property should not be set.

137.10.2.3 public static final String URI_PATH = "org.osgi.rest.uri.path"

This service property describes a URI to the REST extension on this local machine. It is either an fully qualified URI with a different port if no integration with the framework REST service is possible or a relative URI implicitly using the same port if integration is possible. The type of this property is `java.lang.String` and the property is mandatory.

137.11 org.osgi.service.rest.client

Rest Service Client Package Version 1.0.

137.11.1 Summary

- `RestClient` - A Java client API for a REST service endpoint.
- `RestClientFactory` - Factory to construct new REST client instances.

137.11.2 public interface RestClient

A Java client API for a REST service endpoint.

Provides a Java client API for accessing and managing a remote OSGi framework through the REST API. Implementations of this interface will usually take the URL to the remote REST Management Service instance as an argument in their constructor. Further arguments might be needed, for example, if the cloud provider requires URL signing.

Provider Type Consumers of this API must not implement this type

137.11.2.1 public BundleDTO getBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Retrieve the bundle representation for a given bundle Id.

Returns A BundleDTO for the requested bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.2 public BundleDTO getBundle(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Retrieve the bundle representation for a given bundle path.

Returns A BundleDTO for the requested bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.3 public Map<String, String> getBundleHeaders(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the header for a bundle given by its bundle Id.

Returns Returns the map of headers entries.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.4 public Map<String, String> getBundleHeaders(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Get the header for a bundle given by its URI path.

Returns Returns the map of headers entries.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.5 public Collection<String> getBundlePaths() throws Exception

□ Get the bundles currently installed on the managed framework.

Returns Returns a collection of the bundle URIs in the form of Strings. The URIs are relative to the REST API root URL and can be used to retrieve bundle representations.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.6 public Collection<BundleDTO> getBundles() throws Exception

□ Get the bundle representations for all bundles currently installed in the managed framework.

Returns Returns a collection of BundleDTO objects.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.7 public BundleStartLevelDTO getBundleStartLevel(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the start level for a bundle given by its bundle Id.

Returns Returns a BundleStartLevelDTO describing the current start level of the bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.8 public BundleStartLevelDTO getBundleStartLevel(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Get the start level for a bundle given by its URI path.

Returns Returns a BundleStartLevelDTO describing the current start level of the bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.9 public int getBundleState(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the state for a given bundle Id.

Returns Returns the current bundle state as defined in (@link org.osgi.framework.Bundle}.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.10 public int getBundleState(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Get the state for a given bundle path.

Returns Returns the current bundle state as defined in (@link org.osgi.framework.Bundle}.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.11 public FrameworkStartLevelDTO getFrameworkStartLevel() throws Exception

□ Retrieves the current framework start level.

Returns Returns the current framework start level in the form of a FrameworkStartLevelDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.12 public Collection<String> getServicePaths() throws Exception

□ Gets a collection of URI paths to all installed services.

Returns Returns a collection of URI paths to the installed services.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.13 public Collection<String> getServicePaths(String filter) throws Exception

filter Passes a filter to restrict the result set.

□ Gets a collection of URI paths to all installed services.

Returns Returns a collection of URI paths to the installed services.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.14 public ServiceReferenceDTO getServiceReference(long id) throws Exception

id Addresses the service by its identifier.

□ Get the service representation for a service given by its service Id.

Returns The service representation as ServiceReferenceDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.15 public ServiceReferenceDTO getServiceReference(String servicePath) throws Exception

servicePath Addresses the service by its URI path.

□ Get the service representation for a service given by its URI path.

Returns The service representation as ServiceReferenceDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.16 public Collection<ServiceReferenceDTO> getServiceReferences() throws Exception

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.17 public Collection<ServiceReferenceDTO> getServiceReferences(String filter) throws Exception

filter Passes a filter to restrict the result set.

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.18 public BundleDTO installBundle(String location) throws Exception

location Passes the location string to retrieve the bundle content from.

□ Install a new bundle given by an externally reachable location string, typically describing a URL.

Returns Returns the BundleDTO of the newly installed bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.19 public BundleDTO installBundle(String location, InputStream in) throws Exception

location Passes the location string to be used to install the new bundle.

in Passes the input stream to a bundle.

□ Install a new bundle given by an InputStream to a bundle content.

Returns Returns the BundleDTO of the newly installed bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.20 public void setBundleStartLevel(long id, int startLevel) throws Exception

id Addresses the bundle by its identifier.

startLevel The target start level.

□ Set the start level for a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.21 public void setBundleStartLevel(String bundlePath, int startLevel) throws Exception

bundlePath Addresses the bundle by its URI path.

startLevel The target start level.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.22 public void setFrameworkStartLevel(FrameworkStartLevelDTO startLevel) throws Exception

startLevel set the framework start level to this target.

□ Sets the current framework start level.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.23 public void startBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.24 public void startBundle(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Start a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.25 public void startBundle(long id, int options) throws Exception

id Addresses the bundle by its identifier.

options Passes additional options as defined in org.osgi.framework.Bundle.start(int)

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.26 public void startBundle(String bundlePath, int options) throws Exception

bundlePath Addresses the bundle by its URI path.

options Passes additional options as defined in org.osgi.framework.Bundle.start(int)

□ Start a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.27 public void stopBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Stop a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.28 public void stopBundle(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Stop a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.29 public void stopBundle(long id, int options) throws Exception

id Addresses the bundle by its identifier.

options Passes additional options as defined in org.osgi.framework.Bundle.stop(int)

□ Stop a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.30 public void stopBundle(String bundlePath, int options) throws Exception

bundlePath Addresses the bundle by its URI path.

options Passes additional options as defined in org.osgi.framework.Bundle.stop(int)

□ Stop a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.31 public BundleDTO uninstallBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Uninstall a bundle given by its bundle Id.

Returns Returns the BundleDTO of the uninstalled bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.32 public BundleDTO uninstallBundle(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

□ Uninstall a bundle given by its URI path.

Returns Returns the BundleDTO of the uninstalled bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.33 public BundleDTO updateBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Updates a bundle given by its bundle Id using the bundle-internal update location.

Returns Returns the BundleDTO of the updated bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.34 public BundleDTO updateBundle(long id, String url) throws Exception

id Addresses the bundle by its identifier.

url The URL whose content is to be used to update the bundle.

□ Updates a bundle given by its URI path using the content at the specified URL.

Returns Returns the BundleDTO of the updated bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.2.35 public BundleDTO updateBundle(long id, InputStream in) throws Exception

id Addresses the bundle by its identifier.

in Passes an input stream to the new bundle content.

□ Updates a bundle given by its bundle Id and passing the new bundle content in the form of an InputStream.

Returns Returns the BundleDTO of the updated bundle.

Throws `Exception`– An exception representing a failure in the underlying REST call.

137.11.3 public interface RestClientFactory

Factory to construct new REST client instances. Each instance is specific to a REST service endpoint.

Implementations can choose to extend this interface to add additional creation methods, where additional arguments are needed for request signing, etc.

In OSGi environments, this factory is registered as a service.

Provider Type Consumers of this API must not implement this type

137.11.3.1 public RestClient createRestClient(URI uri)

uri The URI to the REST service endpoint.

□ Create a new REST client instance.

Returns A new REST client instance for the specified REST service endpoint.

137.12 JavaScript Client API

REST JavaScript Client API Version 1.0

137.12.1 Summary

- `OSGiRestClient` - A JavaScript client API for accessing and managing a remote OSGi framework through the REST API.
- `OSGiRestCallback` - Callback object provided to the `OSGiRestClient` functions. Invoked on completion of the remote invocation.

JavaScript does not support the concept of interfaces and therefore implementations of the JavaScript client specification can provide objects of any type as long as they conform to the to the signatures described in this specification.

To facilitate documenting the JavaScript APIs *Web IDL* is used; see [2] *Web IDL*. This clarifies the accepted arguments and return types for otherwise untyped functions. Web IDL is only used for documentation purposes and has no bearing on the implementation of this API.

Note: some data types in Web IDL have slightly different names than commonly used in languages like Java or JavaScript. For example a `String` is called `DOMString` and the equivalent of a Java `long` is called `long long`. Additionally, when a representation as defined in this specification is passed to one of the JavaScript client APIs this representation is provided as a JavaScript object. Following the recommendations for mapping these to Web IDL, these JavaScript Object parameters are described using the `dictionary` data type. For more information see the Web IDL specification.

137.12.2 interface OSGiRestClient

Provides a JavaScript client API for accessing and managing a remote OSGi framework through the REST API. Implementations will provide a proprietary constructor to create objects of this signature. Once created the object can be used from JavaScript environments to manage the framework.

137.12.2.1 void getBundle((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle representation as JavaScript object.

□ Get the Bundle representation of a specific bundle.

137.12.2.2 void getBundleHeader((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle Header representation as JavaScript object.

□ Get the Bundle Header representation of a specific bundle.

137.12.2.3 void getBundleRepresentations(OSGiRestCallback cb)

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle Representations List representation as JavaScript object.

□ List the bundles details.

137.12.2.4 void getBundles(OSGiRestCallback cb)

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle List representation as JavaScript object.

□ List the bundles.

137.12.2.5 void getBundleStartLevel((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle Start Level representation as JavaScript object.

□ Get the Bundle Start Level representation of a specific bundle.

137.12.2.6 void getBundleState((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle State representation as JavaScript object.

□ Get the Bundle State representation of a specific bundle.

137.12.2.7 void getFrameworkStartLevel(OSGiRestCallback cb)

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Framework Start Level representation as JavaScript object.

□ Obtain the Framework Start Level.

137.12.2.8 void getService((DOMString or long long) service, OSGiRestCallback cb)

service The service, either the numeric service ID or the service URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Service representation as JavaScript object.

□ Get a service representation.

137.12.2.9 void getServiceRepresentations(OSGiRestCallback cb)

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Service Representations List representation as JavaScript object.

□ Get all services representations.

137.12.2.10 void getServices(OSGiRestCallback cb)

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Service List representation as JavaScript object.

□ Get all services URIs.

137.12.2.11 void installBundle((DOMString or ArrayBuffer) bundle, OSGiRestCallback cb)

bundle The Bundle to install, either represented as a URL or as an ArrayBuffer of

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle representation of the newly installed Bundle. This parameter is optional.

□ Install a bundle from a URI or by value.

137.12.2.12 void setBundleStartLevel((DOMString or long long) bundle, dictionary bsl, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

bsl A Bundle Start Level representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the resulting Framework Start Level representation as JavaScript object. This parameter is optional.

□ Change the Framework Start Level and/or initial bundle start level.

137.12.2.13 void setBundleState((DOMString or long long) bundle, dictionary state, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

state Bundle State representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the resulting Bundle Start Level representation as JavaScript object. This parameter is optional.

□ Change the Bundle Start Level and/or other options defined in the Bundle Start Level representation.

137.12.2.14 void setFrameworkStartLevel(dictionary fwsl, OSGiRestCallback cb)

fwsl Framework Start Level representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the resulting Framework Start Level representation as JavaScript object. This parameter is optional.

□ Change the Framework Start Level and/or initial bundle start level.

137.12.2.15 void startBundle((DOMString or long long) bundle, long options, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

options The options passed to the bundle's start method as a number. This parameter is optional.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

□ Start a bundle.

137.12.2.16 void stopBundle((DOMString or long long) bundle, long options, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

options The options passed to the bundle's start method as a number. This parameter is optional.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

□ Stop a bundle.

137.12.2.17 void uninstallBundle((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle representation of the uninstalled Bundle. This parameter is optional.

□ Uninstall a bundle.

137.12.2.18 void updateBundle((DOMString or long long) bundle, (DOMString or ArrayBuffer) updated, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

updated The Bundle to update, either represented as a URL or as an ArrayBuffer of

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle representation of the updated Bundle. This parameter is optional.

□ Update a bundle from a URI or by value.

137.12.3 callback interface OSGiRestCallback

Objects implementing this signature are provided by users of the OSGiRestClient as callbacks. One of the callback functions is invoked on completion of a REST invocation.

137.12.3.1 void success(object response)

response The result of the invocation. The type of this parameter is depends on the function being invoked. It can be found in the documentation of the function.

□ Called when the invocation completes successfully.

137.12.3.2 void failure(short httpCode, object response)

httpCode The HTTP code returned. If no HTTP code is associated with the failure this parameter is set to -1.

response The failure response.

□ Called when the invocation failed.

137.13 References

[1] *Framework Filter Syntax*
OSGi Core, Chapter 3.2.7 Filter Syntax

[2] *Web IDL*
https://www.w3.org/TR/WebIDL/

[3] *OSGi XML Schemas*
https://docs.osgi.org/xmlns/

[4] *Whiteboard Pattern*
https://docs.osgi.org/whitepaper/whiteboard-pattern/

138 Asynchronous Service Specification

Version 1.0

138.1 Introduction

OSGi Bundles collaborate using loosely coupled services registered in the OSGi service registry. This is a powerful and flexible model, and allows for the dynamic replacement of services at runtime. OSGi services are therefore a very common interaction pattern within OSGi.

As with most Java APIs and Objects, OSGi services are primarily synchronous in operation. This has several benefits; synchronous APIs are typically easier to write and to use than asynchronous ones; synchronous APIs provide immediate feedback; synchronous implementations typically have a less complex threading model.

Asynchronous APIs, however, have different advantages. Asynchronous APIs can reduce bottlenecks by encouraging more effective use of parallelism, improving the responsiveness of the application. In many cases high throughput systems can be written more simply and elegantly using asynchronous programming techniques.

The *Promises Specification* on page 1389 provides powerful primitives for asynchronous programming, including the ability to compose flows in a functional style. There are, however, many existing services that do not use the Promise API. The purpose of the Asynchronous Service is to bridge the gap between these existing, primarily synchronous, services in the OSGi service registry, and asynchronous programming. The Asynchronous Service therefore provides a way to invoke arbitrary OSGi services asynchronously, providing results and failure notifications through the Promise API.

138.1.1 Essentials

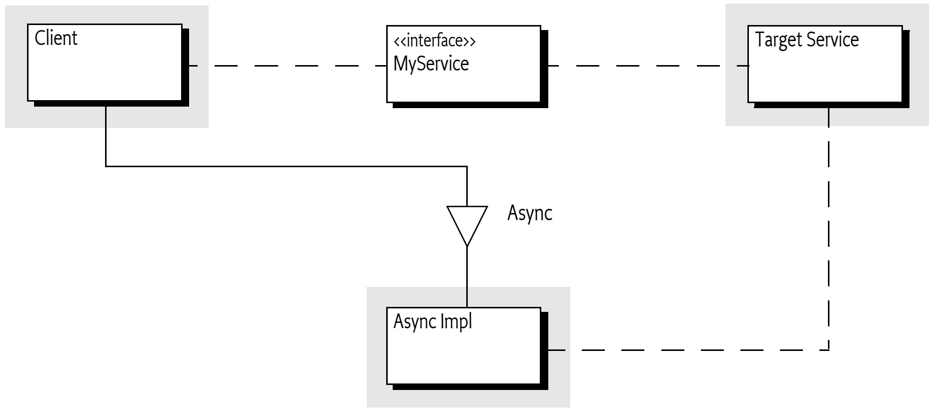
- *Async Invocation* - A single method call that is to be executed without blocking the requesting thread.
- *Client* - Application code that wishes to invoke one or more OSGi services asynchronously.
- *Async Service* - The OSGi service representing the Asynchronous Services implementation. Used by the Client to make one or more Async Invocations.
- *Async Mediator* - A mediator object created by the Async Service which represents the target service. Used by the Client to register Async Invocations.
- *Success Callback* - A callback made when an Async Invocation completes with a normal return value.
- *Failure Callback* - A callback made when an Async Invocation completes with an exception.

138.1.2 Entities

- *Async Service* - A service that can create Async Mediators and run Async Invocations.
- *Target Service* - A service that is to be called asynchronously by the Client.
- *Client* - The code that makes Async Invocations using the Async Service

- *Promise* - A promise, representing the result of the Async Invocation.

Figure 138.1 *Class and Service overview*



138.2 Usage

This section is an introduction in the usage of the Async Service. It is not the formal specification, the normative part starts at *Async Service* on page 751. This section leaves out some of the details for clarity.

138.2.1 Synopsis

The Async Service provides a mechanism for a client to *asynchronously* invoke methods on a target service. The service may be aware of the asynchronous nature of the call and actively participate in it, or be unaware and execute normally. In either case the client's thread will not block, and will continue executing its next instructions. Clients are notified of the completion of their task, and whether it was successful or not, through the use of the Promise API.

Each async invocation is registered by the client making a method call on an *Async Mediator*, and then started by making a call to the Async Service that created the mediator. This call returns a Promise that will eventually be resolved with the return value from the async invocation.

An Async Mediator can be created by the client, either from an Object, or directly from a Service Reference. Using a service reference has the advantage that the mediator will track the underlying service. This means that if the service is unregistered before the asynchronous call begins then the Promise will resolve with a failure, rather than continuing using an invalid service object.

138.2.2 Making Async Invocations

The general pattern for a client is to obtain the Async Service, and a service reference for the target service. The client then creates an Async Mediator for the target service, invokes a method on the mediator, then starts the asynchronous call. This is demonstrated in the following example:

```
private Async asyncService;
private ServiceReference<Foo> fooRef;
private Foo mediated;

@Reference
void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = Foo.class)
void setList(ServiceReference<Foo> foo) {
    fooRef = foo;
}

@Activate
void start() {
    mediated = asyncService.mediate(fooRef, Foo.class);
}

public synchronized void doStuff() {
    Promise<Boolean> promise = asyncService
           .call(mediated.booleanMethod("aValue"));
    ...
}
```

This example demonstrates how simply clients can make asynchronous calls using the Async Service. The eventual result can be obtained from the promise using one of the relevant callbacks.

One important thing to note is that whilst the call to call() or call(R) causes the async invocation to begin, the actual execution of the underlying task may be queued until a thread is available to run it. If the service has been unregistered before the execution actually begins then the promise will be resolved with a Service Exception. The type of the Service Exception will be ASYNC_ERROR.

138.2.3 Async Invocations of Void Methods

The return value of the mediator method call is used to provide type information to the Async Service. This, however, does not work for void methods that have no return value. In this case the client can either pass an arbitrary object to the call(R) method, or use the zero argument call() method. In either case the returned promise will eventually resolve with a value of null. This is demonstrated in the following example.

```
private Async asyncService;
private ServiceReference<Foo> fooRef;
private Foo mediated;

@Reference
void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = Foo.class)
void setList(ServiceReference<Foo> foo) {
    fooRef = foo;
}

@Activate
void start() {
    mediated = asyncService.mediate(fooRef, Foo.class);
}
```

```
public synchronized void doStuff() {
    mediated.voidMethod();
    Promise<?> promise = asyncService
           .call();
    ...
}
```

138.2.4 Fire and Forget Calls

Sometimes a client does not require any notification that an async invocation has completed. In this case the client could use one of the call() or call(R) methods and simply discard the returned Promise object. This, however, can be wasteful of resources. The act of resolving the Promise object may be expensive, for example it may involve serializing the return value over a network if the remote call was asynchronous.

If the client knows that no Promise object representing the result of the asynchronous task is needed then it can signal this to the Async Service. This allows the Async Service to better optimize the async invocation by not providing a result.

To indicate that the client wants to make a fire-and-forget style call the client invokes the mediator as normal, but then begins the asynchronous invocation using the execute() method as show below.

```
private Async asyncService;
private ServiceReference<Foo> fooRef;

private Foo mediated;

@Reference
void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = Foo.class)
void setList(ServiceReference<Foo> foo) {
    fooRef = foo;
}

@Activate
void start() {
    mediated = asyncService.mediate(fooRef, Foo.class);
}

public synchronized void doStuff() {
    mediated.someMethod();
    asyncService.execute();
    ...
}
```

Note that the execute() method does still return a Promise. This Promise is not the same as the ones returned by call() or call(R), its resolution value does not provide access to the result, but instead indicates whether the fire-and-forget call could be successfully started. If there is a failure which prevents the task from being executed then this is used to fail the returned promise.

138.2.5 Multi Threading

By their very definition asynchronous tasks do not run inline, and typically they will not run on the same thread as the caller. This is not, however, a guarantee. A valid implementation of the Async Service may have only one worker thread, which may be the thread currently running in the client code. Async invocations also have the same threading model as the Promise API. This means that callbacks may run on arbitrary threads, which may, or may not, be the same as the client thread, or the thread which executed the asynchronous work.

It is important for multi-threaded clients to note that calls to the mediator and Async Service must occur on the same thread. For example it is not supported to invoke a mediator using one thread, and then to begin the async invocation by calling the `call()`, `call(R)` or `execute()` method on a different thread.

138.3 Async Service

The Async Service is the primary interaction point between a client and the Async Service implementation. An Async Service implementation must expose a service implementing the `Async` interface. Clients obtain an instance of the Async Service using the normal OSGi service registry mechanisms, either directly using the OSGi framework API, or using dependency injection.

The Async Service is used to:

- Create async mediators
- Begin async invocations
- Obtain Promise objects representing the result of the async invocation

138.3.1 Using the Async Service

The first action that a client wishing to make an async invocation must take is to create an async mediator using one of the `mediate` methods. Once created the client invokes the method that should be run asynchronously, supplying the arguments that should be used. This call records the invocation, but does not start the asynchronous task. The asynchronous task begins when the client invokes one of the `call` or `execute` methods on the Async Service. The `call` methods must return a Promise representing the async invocation. The promise must resolve with the value returned by the async invocation, or fail with the failure thrown by the async invocation.

If the client attempts to begin an async invocation without first having called a method on the mediator object then the Async Service must detect this usage error and throw an `IllegalStateException` to the client. This applies to all methods that begin an async invocation.

138.3.2 Asynchronous Failures

There are a variety of reasons that async invocations may be started correctly by the client, but then fail without running the asynchronous task. In any of these cases the Promise representing the async invocation must fail with a Service Exception. This Service Exception must be initialized with a type of `ASYNC_ERROR`. If there is no promise representing the async invocation then there is no way to notify the client of the failure, therefore the Service Exception must be logged by the Async Service using all available Log Service implementations.

The following list of scenarios is not exhaustive, but indicates failure scenarios that must result in a Service Exception with a type of async

- If the client is using a service reference backed mediator and the client bundle's bundle context becomes invalid before looking up the target service.
- If the client is using a service reference backed mediator and the service is unregistered before making the async invocation.

- If the client is using a service reference backed mediator and the service lookup returns null
- If the Async Service is unable to accept new work, for example it is in the process of being shut down.
- If the type of the mediator object does not match the type of the service object to be invoked.

138.3.3 Thread Safety and Instance Sharing

Implementations of the Async Service must be thread safe and may be used simultaneously across multiple clients and from multiple threads within the same client. Whilst the Async Service is able to be used across multiple threads, if a client wishes to make an async invocation then the call to the mediator and the call to begin the async invocation must occur on the same thread. The returned Promise may then be shared between threads if required.

It is expected, although not required, that the Async Service implementation will use a Service Factory to create customized implementations for each client bundle. This simplifies the tracking of the relevant client bundle context to use when performing service lookups on the client bundle's behalf. Clients should therefore not share instances of the Async Service with other bundles. Instead both bundles should obtain their own instances from the service registry.

138.3.4 Service Object Lifecycle Management

If the Async Service is being used to call an OSGi service object and the service reference is available then the service object should be looked up immediately before the asynchronous task begins executing. This ensures that the service is still available at the point it is eventually called. Any call to getService must have a corresponding call to ungetService after the mediated method invoked has returned and, if available, the promise is resolved, but before the asynchronous task releases its thread of execution.

138.4 The Async Mediator

Async mediators are dynamically created objects that have the same type or interface as the object being mediated, and are used to record method invocations and arguments. Mediator objects are specific to an Async Service implementation, and must only be used in conjunction with the Async Service object that they were created by.

Mediators may be created either from a ServiceReference or from a service object. The actions and overall result are similar for both the mediate(ServiceReference,Class) and mediate(T,Class) methods, with the primary difference being that mediated objects created from a ServiceReference will validate whether the service object is still available immediately before the asynchronous task is executed.

138.4.1 Building the Mediator Object

The client passes in a Class indicating the type that should be mediated. If the class object represents an interface type then the generated mediator object must implement that interface. If the class object represents a Java class type then the mediator object must either be an instance of that type or extend it.

When building a mediator object the Async Service has the opportunity to detect numerous problems, for example if the referenced service to be mediated has been unregistered. Although fail-fast behavior is usually preferable, in this case it would force the client to handle errors in two places; both when creating the mediator, and for the returned Promise. To simplify client usage, error cases detected when creating a mediator must not prevent the mediator from being created and must not result in an exception being thrown. The only reason that the Async Service may fail to create a mediator is if the class object passed in cannot be mediated.

There are three reasons why the Async Service may not be able to mediate a class type:

- The class object passed in represents a final type.
- The class object passed in represents a type that has no zero-argument constructor.
- The class object passed in represents a type which has one or more public final methods present in its type hierarchy (other than those declared by `java.lang.Object`).

If any of these constraints are violated and prevent the Async Service from creating a mediator then the Async Service must throw an IllegalArgumentException.

138.4.2 Async Mediator Behaviors

When invoked, the Async mediator must record the method call, and its arguments, and then return rapidly and should avoid performing blocking operations. The values returned by the mediator object are opaque, and the client should not attempt to interpret the returned value. The value may be null (or null-like in the case of primitives) or contain implementation specific information. If the mediated method call has a return type, specifically it is non-void, then this object must be passed to the Async Service's `call` method when beginning the async invocation

Async mediators should make a best-effort attempt to detect incorrect API usage from the client. If this incorrect usage is detected then the mediator object must throw an IllegalStateException when invoked. An example of incorrect usage that must be detected is when a client makes multiple invocations on a single mediator object from the same thread without making any calls to the Async Service.

After a usage error has been detected and an IllegalStateException has been thrown the mediator object must be reset so that a subsequent invocation from the client thread can proceed normally.

138.4.3 Thread Safety and Instance Sharing

Async mediators, like instances of the Async Service, are required to be thread safe. Clients may therefore share mediator objects across threads, and can safely store them as instance fields. Whilst mediators are thread safe, if a client wishes to make an async invocation then the call to the mediator and the call to `call()` or `call(R)` must occur on the same thread. The returned Promise may then be shared between threads if required.

Async mediators created from `ServiceReference` objects remain directly associated with the service reference and client bundle after creation. Clients should therefore not share mediator objects with other bundles. Instead each bundle should create its own mediator.

138.5 Fire and Forget Invocations

The Async Service provides `call()` and `call(R)` methods for clients to use when they wish to receive results from asynchronous tasks. Clients that do not need the result can simply discard the returned Promise object. This, however, can be wasteful of resources. The act of resolving the Promise object may be expensive, for example it may involve serializing the return value over a network.

To address this use case the Async Service provides the `execute()` method, which behaves similarly to `call()` and `call(R)`, but does not provide access to the eventual result. Instead the `execute()` method returns a `Promise` that indicates whether the fire-and-forget call is able to be successfully started.

The returned Promise must be resolved with `null` if the asynchronous task begins executing successfully. There is no *happens-before* relationship required, meaning that if the Promise resolves successfully then the task may, or may not, have started or finished. The primary usage of the Promise is actually to detect failures. If the fire-and-forget task cannot be executed for some reason, for example the backing service has been unregistered, then the returned promise must be failed appropriately using the same rules as defined in *Asynchronous Failures* on page 751. If the returned Promise is failed then the fire-and-forget task has not executed and will not execute in the future.

138.6 Delegating to Asynchronous Implementations

Some service APIs are already asynchronous in operation, and others are partly asynchronous, in that some methods run asynchronously and others do not. There are also services which have a synchronous API, but could run asynchronously because they are a proxy to another service. A good example of this kind of service is a remote service. Remote services are local views of a remote endpoint, and depending upon the implementation of the endpoint it may be possible to make the remote call asynchronously, optimizing the thread usage of any local asynchronous call.

Services that already have some level of asynchronous support may advertise this to clients and to the Async Service by having their service object be an `instanceof` `AsyncDelegate`. The service object can be cast to `AsyncDelegate` to be used by the Async Service implementation, or by the client directly, to make an asynchronous call on the service.

Because the Async Delegate behavior is transparently handled by the Async Service, clients of the Async Service do not need to know whether the service object is an `instanceof` `AsyncDelegate` or not. Their usage pattern can remain unchanged.

When making an async invocation, the Async Service must check to see whether the service object is an `instanceof` `AsyncDelegate`. If the service object is an `instanceof` `AsyncDelegate`, then the Async Service must attempt to delegate the asynchronous call. The exact delegation operation depends on whether a Promise result is required.

138.6.1 Obtaining a Promise from an Async Delegate

If the result of the method invocation is needed by the client, then the Async Service must attempt to delegate to the `async(Method,Object[])` method. The delegation proceeds as follows:

- If the call to the Async Delegate returns a Promise, then the Promise returned by the Async Service must be resolved with that Promise.
- If the call to the Async Delegate throws an exception, then the Promise returned by the Async Service must be failed with the exception.
- If the Async Delegate is unable to optimize the call and returns `null` from the `async(Method,Object[])` method, the Async Service must continue processing the async invocation, treating the service as a normal service object.

138.6.2 Delegating Fire and Forget Calls to an Async Delegate

If the result of the method invocation is not needed by the client, then the Async Service must attempt to delegate to the `execute(Method,Object[])` method. This gives the Async Delegate implementation the opportunity to further optimize its processing. The delegation proceeds as follows:

- If the call to the Async Delegate returns `true`, then the Promise returned by the Async Service must be resolved with `null`.
- If the call to the Async Delegate throws an exception, then the Promise returned by the Async Service must be failed with the exception.
- If the Async Delegate is unable to optimize the call and returns `false` from the `execute(Method,Object[])` method, the Async Service must continue processing the async invocation, treating the service as a normal service object.

138.6.3 Lifecycle for Service Objects When Delegating

If an Async Delegate implementation accepts an asynchronous task, via a call to either `execute(Method,Object[])` or `async(Method,Object[])`, then it is responsible for continuing to process the work until completion. This means that if the service implementing Async Delegate is unregistered for some reason, then the task must be properly cleaned up and succeed or fail as appropriate.

If the Async Service implementation used a service reference to obtain the service, then it must release the service object after the task has been accepted. This means that if the service object is provided by a service factory, then the service object should take extra care not to destroy its internal state when released. The service object must remain valid until all executing asynchronous tasks associated with the service object are either completed or failed.

If an Async Delegate implementation rejects an asynchronous task, by returning false or null, the Async Service implementation must take over the asynchronous invocation of the method. In this case, if the Async Service implementation used a service reference to obtain the service, the Async Service must not release the service object until the asynchronous task is completed.

If an Async Delegate implementation throws an exception and the Async Service implementation used a service reference to obtain the service, then the service object must be released immediately.

138.7 Capabilities

Implementations of the Asynchronous Service specification must provide the following capabilities.

- A capability in the osgi.implementation namespace declaring the implemented specification to be osgi.async. This capability must also declare a uses constraint for the org.osgi.service.async and org.osgi.service.async.delegate packages. For example:

  ```
  Provide-Capability: osgi.implementation;
      osgi.implementation="osgi.async";
      version:Version="1.0";
      uses:="org.osgi.service.async,org.osgi.service.async.delegate"
  ```

  This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.
- A capability in the osgi.service namespace representing the Async service. This capability must also declare a uses constraint for the org.osgi.service.async package. For example:

  ```
  Provide-Capability: osgi.service;
      objectClass:List<String>="org.osgi.service.async.Async";
      uses:="org.osgi.service.async"
  ```

  This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

138.8 Security

Asynchronous Services implementations must be careful to avoid elevating the privileges of client bundles when calling services asynchronously, and also to avoid restricting the privileges of clients that are permitted to make a call. This means that the implementation must:

- Be granted AllPermission. As the Async Service will always be on the stack when invoking a service object asynchronously it must be granted AllPermission so that it does not interfere with security any checks made by the service object.
- Establish the caller's AccessControlContext in a worker thread before starting to call the service object. This prevents a bundle from being able to call a service asynchronously that it would not normally be able to call. The AccessControlContext must be collected during any call to call(), call(R) or execute().
- Use a doPrivileged block when mediating a concrete type. A no-args constructor in a concrete type may perform actions that the client may not have permission to perform. This should not

prevent the client from mediating the object, as the client is not directly performing these actions.
- If the mediator object was created using a service reference, then the Async Services implementation must use the client's bundle context when retrieving the target service. If the service lookup occurs on a worker thread, then the lookup must use the `AccessControlContext` collected during the call to `call()`, `call(R)` or `execute()`. This prevents the client bundle from being able to make calls on a service object that they do not have permission to obtain, and ensures that an appropriately customized object is returned if the service is implemented using a service factory.

Further security considerations can be addressed using normal OSGi security rules. For example access to the Async Service can be controlled using `ServicePermission[...Async, GET]`.

138.9 org.osgi.service.async

Asynchronous Services Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.async; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.async; version="[1.0,1.1)"`

138.9.1 Summary

- `Async` - The Asynchronous Execution Service.

138.9.2 public interface Async

The Asynchronous Execution Service. This can be used to make asynchronous invocations on OSGi services and objects through the use of a mediator object.

Typical usage:

```
Async async = ctx.getService(asyncRef);

ServiceReference<MyService> ref = ctx.getServiceReference(MyService.class);

MyService mediator = async.mediate(ref, MyService.class);

Promise<BigInteger> result = async.call(mediator.getSumOverAllValues());
```

The Promise API allows callbacks to be made when asynchronous tasks complete, and can be used to chain Promises.

Multiple asynchronous tasks can be started concurrently, and will run in parallel if the Async Service has threads available.

Provider Type Consumers of this API must not implement this type

138.9.2.1 public Promise<R> call(R r)

Type Parameters `<R>`

r The return value of the mediated call, used for type information.

□ Invoke the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned Promise.

Typically the parameter for this method will be supplied inline like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A Promise which can be used to retrieve the result of the asynchronous task.

138.9.2.2 public Promise<?> call()

□ Invoke the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned Promise.

Generally it is preferable to use call(Object) like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

However this pattern does not work for void methods. Void methods can therefore be handled like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
i.voidMethod()
Promise<?> p = async.call();
```

Returns A Promise which can be used to retrieve the result of the asynchronous task.

138.9.2.3 public Promise<Void> execute()

□ Invoke the last method call registered by a mediated object as a "fire-and-forget" asynchronous task. This method should be used by clients in preference to call() and call(Object) when no callbacks, or other features of Promise, are needed.

The advantage of this method is that it allows for greater optimization of the underlying asynchronous task. Clients are therefore likely to see better performance when using this method compared to using call(Object) or call() and ignoring the returned Promise. The Promise returned by this method is different from the Promise returned by call(Object) or call(), in that the returned Promise will resolve when the fire-and-forget task is successfully started, or fail if the task cannot be started. Note that there is no *happens-before* relationship and the returned Promise may resolve before or after the fire-and-forget task starts, or completes.

Typically this method is used like call():

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
i.someMethod()
Promise<Void> p = async.execute();
```

Returns A Promise representing whether the fire-and-forget task was able to start.

138.9.2.4 public T mediate(T target, Class<T> iface)

Type Parameters <T>

target The service object to mediate.

iface The type that the mediated object should provide.

□ Create a mediator for the specified object. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the call(Object), call(), or execute() method.

The values returned by method calls made on a mediated object are opaque and should not be interpreted.

Normal usage:

```
I s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A mediator for the service object.

Throws IllegalArgumentException– If the type represented by iface cannot be mediated.

138.9.2.5 public T mediate(ServiceReference<? extends T> target, Class<T> iface)

Type Parameters <T>

target The service reference to mediate.

iface The type that the mediated object should provide.

□ Create a mediator for the specified service. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the call(Object), call(), or execute() method.

The values returned by method calls made on a mediated object are opaque and should not be interpreted.

This method differs from mediate(Object, Class) in that it can track the availability of the specified service. This is recommended as the preferred option for mediating OSGi services as asynchronous tasks may not start executing until some time after they are requested. Tracking the validity of the ServiceReference for the service ensures that these tasks do not proceed with an invalid object.

Normal usage:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A mediator for the service object.

Throws IllegalArgumentException– If the type represented by iface cannot be mediated.

138.10 org.osgi.service.async.delegate

Asynchronous Services Delegation Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package contains only interfaces that are implemented by consumers.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.async.delegate; version="[1.0,2.0)"

138.10.1 Summary

- AsyncDelegate - This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.

138.10.2 public interface AsyncDelegate

This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.

This may mean that the service has access to its own thread pool, or that it can delegate work to a remote node, or act in some other way to reduce the load on the Asynchronous Services implementation when making an asynchronous call.

138.10.2.1 public Promise<?> async(Method m, Object[] args) throws Exception

m The method to be asynchronously invoked.

args The arguments to be used to invoke the method.

□ Invoke the specified method as an asynchronous task with the specified arguments.

This method can be used by clients, or the Async Service, to optimize Asynchronous execution of methods.

When called, this method should invoke the supplied method using the supplied arguments asynchronously, returning a Promise that can be used to access the result.

If the method cannot be executed asynchronously by this method then `null` must be returned.

Returns A Promise representing the asynchronous result, or `null` if this method cannot be asynchronously invoked.

Throws `Exception`– An exception should be thrown only if there was a serious error that prevented the asynchronous task from starting. For example, the specified method does not exist on this object. Exceptions must not be thrown to indicate that the call does not support asynchronous invocation. Instead this method must return `null`. Exceptions must also not be thrown to indicate a failure from the execution of the underlying method. This must be handled by failing the returned Promise.

138.10.2.2 public boolean execute(Method m, Object[] args) throws Exception

m The method to be asynchronously invoked.

args The arguments to be used to invoke the method.

□ Invoke the specified method as a "fire-and-forget" asynchronous task with the specified arguments.

This method can be used by clients, or the Async Service, to optimize Asynchronous execution of methods.

When called, this method should invoke the specified method using the specified arguments asynchronously. This method differs from async(Method, Object[]) in that it does not return a Promise. This method therefore allows the implementation to perform more aggressive optimizations because the end result of the invocation does not need to be returned to the caller.

If the method cannot be executed asynchronously by this method then `false` must be returned.

Returns `true` if the asynchronous execution request has been accepted, or `false` if this method cannot be asynchronously invoked by the AsyncDelegate.

Throws `Exception`– An exception should be thrown only if there was a serious error that prevented the asynchronous task from starting. For example, the specified method does not exist on this object. Exceptions must not be thrown to indicate that the call does not support asynchronous invocation. Instead this method must return `false`. Exceptions must also not be thrown to indicate a failure from the execution of the underlying method.

139 Device Service Specification for EnOcean™ Technology

Version 1.0

139.1 Introduction

EnOcean is a standard wireless communication protocol designed for low-cost and low-power devices by EnOcean Alliance.

EnOcean is widely supported by various types of devices such as smart meters, lights and many kinds of sensors in the residential area. OSGi applications need to communicate with those EnOcean devices. This specification defines how OSGi bundles can be developed to discover and control EnOcean devices on the one hand, and act as EnOcean devices and interoperate with EnOcean clients on the other hand. In particular, a Java mapping is provided for the standard representation of EnOcean devices called EnOcean Equipment Profile (EEPs). See [2] *EnOcean Equipment Profiles v2.6.2*.

The specification also describes the external API of an EnOcean Base Driver according to Device Access specification.

139.2 Essentials

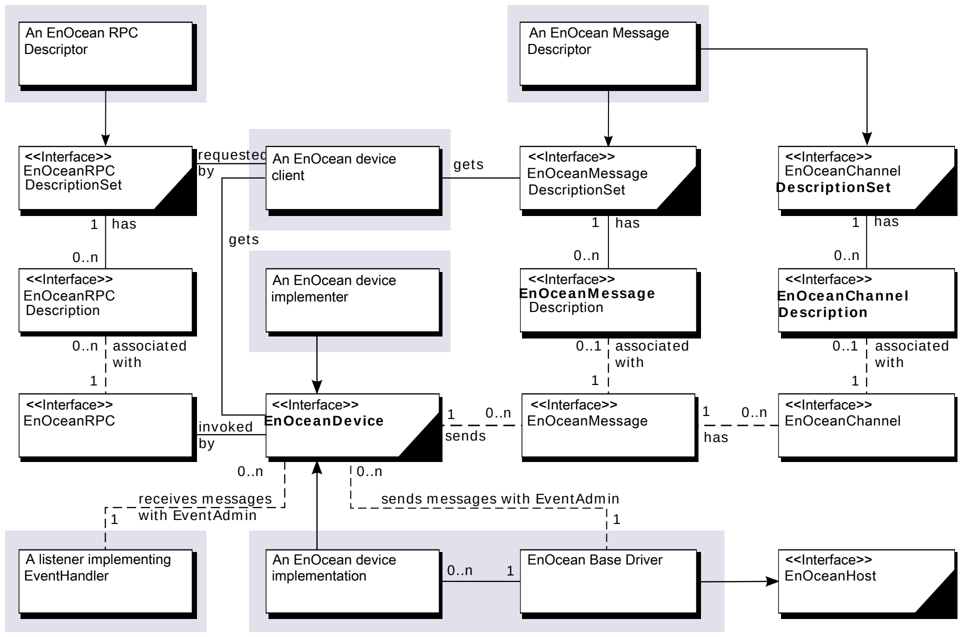
- *Scope* - This specification is limited to general device discovery and control aspects of the standard EnOcean specifications. Aspects concerning the representation of specific or proprietary EnOcean profiles is not addressed.
- *Transparency* - EnOcean devices discovered on the network and devices locally implemented on the platform are represented in the OSGi service registry with the same API.
- *Lightweight implementation option* - The full description of EnOcean device services on the OSGi platform is optional. Some base driver implementations may implement all the classes including EnOcean device description classes while Implementations targeting constrained devices are able to implement only the part that is necessary for EnOcean device discovery and control.
- *Network Selection* - It must be possible to restrict the use of the EnOcean protocols to a selection of the connected devices.
- *Event handling* - Bundles are able to listen to EnOcean events.
- *Discover and control EnOcean devices as OSGi services* - Available learned (via an EnOcean teach-in procedure) EnOcean external endpoints are dynamically reified as OSGi services on the service registry upon discovery.
- *OSGi services as exported EnOcean devices* - OSGi services implementing the API defined here and explicitly set to be exported should be made available to networks with EnOcean-enabled endpoints in a transparent way.

139.3 Entities

- *EnOcean Base Driver* - The bundle that implements the bridge between OSGi and EnOcean networks, see Figure 139.1 on page 763. It is responsible for accessing the various EnOcean gateway chips on the execution machine, and ensures the reception and translation of EnOcean messages into proper objects. It is also used to send messages on the EnOcean network, using whatever chip it deems most appropriate.
- *EnOcean Host* - The EnOceanHost object is a link between the software and the EnOcean network. It represents the chip configuration (gateway capabilities) described in [5] *EnOcean System Specification - Security of EnOcean Radio Networks v1.9*. It is registered as an OSGi service.
- *EnOcean Device* - An EnOcean device. This entity is represented by a EnOceanDevice interface and registered as a service within the framework. It carries the unique chip ID of the device, and may represent either an imported or exported device, which may be a pure transmitter or a transceiver.
- *EnOcean Message* - Every EnOcean reporting equipment is supposed to follow a "profile", which is essentially the way the emitted data is encoded. In order to reflect this standard as it is defined in [2] *EnOcean Equipment Profiles v2.6.2*, manufacturers are able to register the description of "Messages", the essence of a profile, along with their associated payload (as Channels). See "EnOcean Channels" below for more information.
- *EnOcean Channel* - EnOcean channels are available as an array inside EnOceanMessage objects. They are a useful way to define any kind of payload that would be put inside of an EnOcean Message.

  EnOcean Messages and their associated Channels can be described with EnOceanMessageDescription and EnOceanChannelDescription interfaces. Description providers aggregate these descriptions in sets that they register with EnOceanMessageDescriptionSet and EnOceanChannelDescriptionSet interfaces within the framework.
- *EnOcean RPC* - An interface that enables the invocation of vendor-specific Remote Procedure Calls and Remote Management Commands. These are particular types of Messages and are not linked to any EnOcean Profile, so that their descriptions are defined and registered in another way. The RPCs are documented via the EnOceanRPCDescription objects gathered into registered EnOceanRPCDescriptionSet services.
- *EnOcean Handler* - Enables clients to asynchronously get answers to their RPCs.
- *EnOcean Client* - An application that is intended to control EnOcean device services.
- *EnOcean Exception* - Delivers errors during EnOceanMessage serialization/deserialization or during execution outside transmission.

Figure 139.1 *EnOcean Service Specification class diagram.*



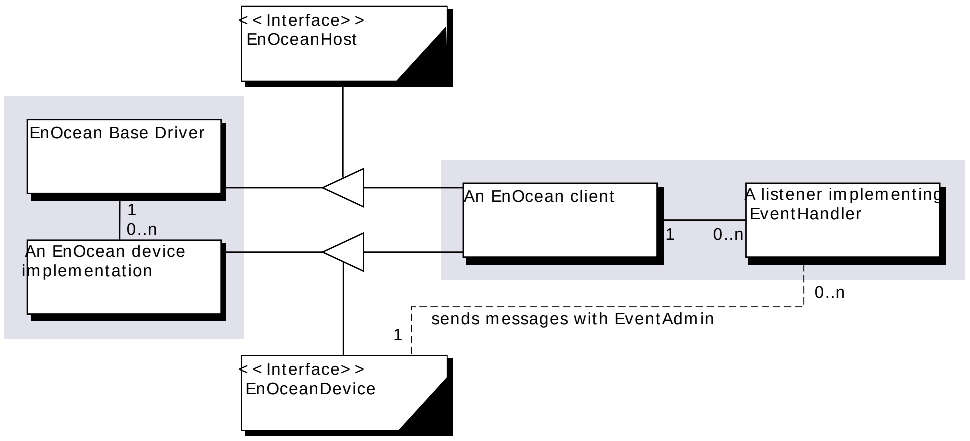
An EnOcean RPC Descriptor
An EnOcean Message Descriptor
<<Interface>> EnOceanRPC DescriptionSet
requested by
An EnOcean device client
gets
<<Interface>> EnOceanMessage DescriptionSet
<<Interface>> EnOceanChannel DescriptionSet
1 has 0..n
gets
<<Interface>> EnOceanRPC Description
An EnOcean device implementer
<<Interface>> EnOceanMessage Description
<<Interface>> EnOceanChannel Description
0..n associated with 1
0..1 associated with 1
<<Interface>> EnOceanRPC
invoked by
<<Interface>> EnOceanDevice
1 0..n sends
<<Interface>> EnOceanMessage
1 0..n has
<<Interface>> EnOceanChannel
receives messages with EventAdmin
sends messages with EventAdmin
A listener implementing EventHandler
An EnOcean device implementation
EnOcean Base Driver
<<Interface>> EnOceanHost

139.4 Operation Summary

To make an EnOcean device service available to EnOcean clients on the OSGi platform, it must be registered under the EnOceanDevice interface within the OSGi framework.

The EnOcean Base Driver is responsible for mapping external devices into EnOceanDevice objects, through the use of an EnOcean gateway. See [1] *Pervasive Service Composition in the Home Network*. The latter is represented on OSGi framework as an object implementing EnOceanHost interface. EnOcean "teach-in" messages will trigger this behavior, this is called a device import situation, see Figure 139.2 on page 763.

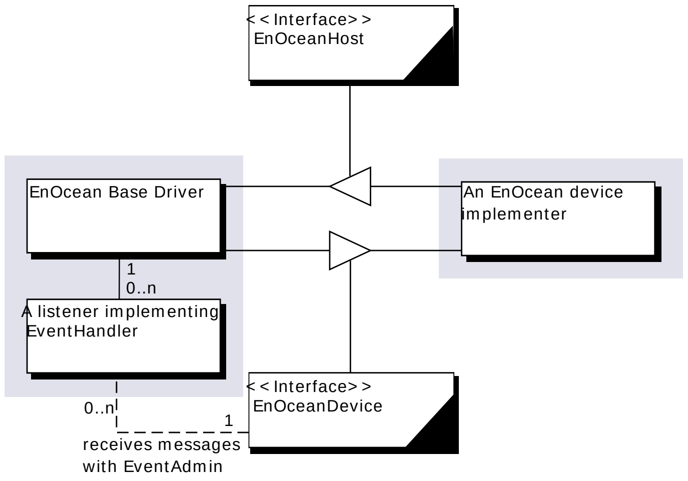
Figure 139.2 *EnOcean device import.*



<<Interface>> EnOceanHost
EnOcean Base Driver
1 0..n
An EnOcean device implementation
An EnOcean client
1 0..n
A listener implementing EventHandler
0..n
sends messages with EventAdmin
1
<<Interface>> EnOceanDevice

Client bundles may also expose framework-internal (local) EnOceanDevice instances, registered within the framework, see Figure 139.3 on page 764. The Base Driver then should emulate those objects as EnOcean devices on the EnOcean network. This is a device export situation, made possible by the use of the 127 virtual base IDs available on an EnOcean gateway. For more information about this process, see *Export Situation* on page 766.

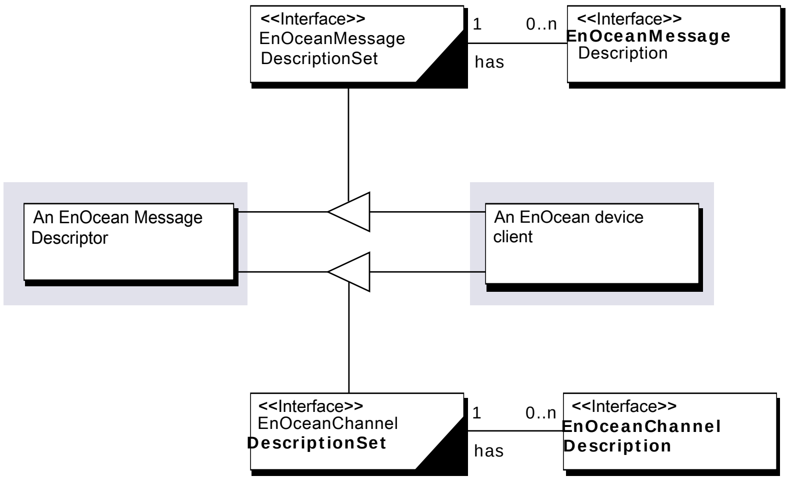
Figure 139.3 *EnOcean device export.*


<< Interface>>
EnOceanHost
EnOcean Base Driver
An EnOcean device implementer
1
0..n
A listener implementing EventHandler
<< Interface>>
EnOceanDevice
0..n
1
receives messages with EventAdmin

EnOcean clients send RPCs (Remote Procedure Calls) to EnOcean devices and receives RPC responses and messages from them. Messages coming from EnOcean devices are accessible through Event Admin.

RPCs and messages content are specified by EnOcean Alliance or vendor-specific descriptions. Those descriptions may be provided on the OSGi platform by any bundle through the registration of EnOceanRPCDescriptionSet, EnOceanMessageDescriptionSet and EnOceanChannelDescriptionSet services. Every service is a set of description that enables applications to retrieve information about supported RPCs, messages or channels that compose messages.

Figure 139.4 *Using a set of message descriptions.*


<<Interface>>
EnOceanMessage DescriptionSet
1
0..n
has
<<Interface>>
EnOceanMessage Description
An EnOcean Message Descriptor
An EnOcean device client
<<Interface>>
EnOceanChannel DescriptionSet
1
0..n
has
<<Interface>>
EnOceanChannel Description

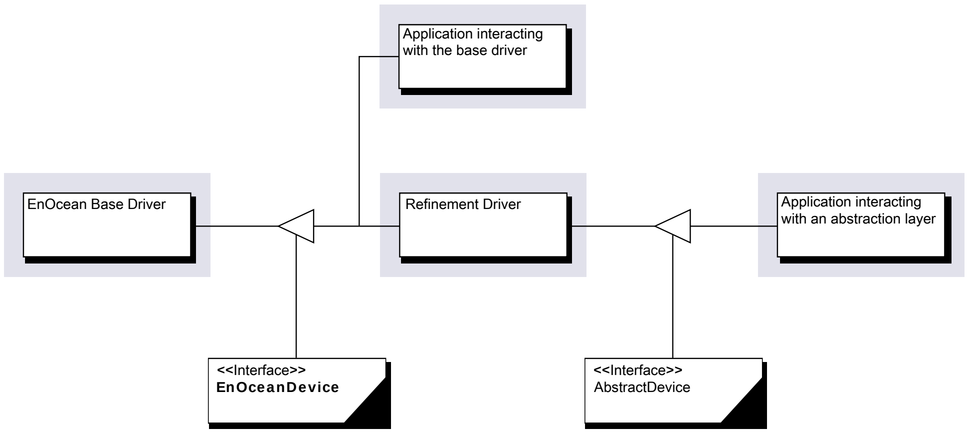
139.5 EnOcean Base Driver

Most of the functionality described in the operation summary is implemented in an EnOcean base driver. This bundle implements the EnOcean protocol and handles the interaction with bundles that use the EnOcean devices. An EnOcean base driver is able to discover EnOcean devices on the network and map each discovered device into an OSGi registered EnOceanDevice service. It is also the receptor, through EventAdmin service and OSGi service registry, of all the events related to local devices and clients. It enables bidirectional communication for RPC and Channel updates.

Several base drivers may be deployed on a residential OSGi device, one for every supported network technology. An OSGi device abstraction layer may then be implemented as a layer of refinement drivers above a layer of base drivers. The refinement driver is responsible for adapting technology-specific device services registered by the base driver into device services of another model, see AbstractDevice interface in Figure 139.5 on page 765. In the case of a generic device abstraction layer, the model is agnostic to technologies.

The EnOcean Alliance defines their own abstract model with EnOcean Equipment Profiles and refinement drivers may provide the implementation of all EEPs with EnOcean specific Java interfaces. The AbstractDevice interface of Figure 139.5 on page 765 is then replaced by an EEP specific Java interface in that case. The need and the choice of the abstraction depends on the targeted application domain.

Figure 139.5 *EnOcean Base Driver and a refinement driver representing devices in an abstract model.*


Application interacting with the base driver
EnOcean Base Driver
Refinement Driver
Application interacting with an abstraction layer
<<Interface>> EnOceanDevice
<<Interface>> AbstractDevice

139.6 EnOcean Host

The EnOcean host represents an EnOcean gateway chip. Any EnOcean device service implementation should rely on at least one Gateway Chip in order to send and receive messages on the external EnOcean network. This interface enables standard control over an EnOcean compatible chip. Every EnOceanHost object should at least be identified by its unique chip ID.

The EnOceanHost interface enables OSGi applications to:

- Get or set gateway metadata (version, name, etc);
- Reset the gateway chip device;
- Retrieve a chip ID (derived from EnOcean's BASE_ID) for the given Service PID of a device.

139.7 EnOcean Device

139.7.1 Generics

A physical EnOcean device is reified as an EnOceanDevice object within the framework.

An EnOcean device holds most of the natural properties for an EnOcean object: its unique ID, the profile, a friendly name, its security information, and its available RPCs – along with the associated getters (and setters when applicable). All those properties MUST be persistent across restart so that teach-in procedures are made only once.

It also holds methods that reflect the natural actions a user application may physically trigger on such a device: send a message to the device, send a teach-in message to the device, or switch the device to learning mode.

Every EnOcean Device keeps a service PID property that is assigned either by the base driver or by any service-exporting bundle. The property value format is free and the value must be unique on the framework.

The properties on which EnOceanDevice services can be filtered on are: the device's service PID and chip ID, and its profile identifiers (RORG / FUNC / TYPE integers).

The EnOceanDevice also keeps security features as defined in the EnOcean Security Draft, [5] *EnOcean System Specification - Security of EnOcean Radio Networks v1.9*, which allow for a security level format (integer mask), an encryption key and/or a rolling authentication code.

The EnOceanDevice service MUST also be registered with the DEVICE_CATEGORY service property, see *Device Service Registration* on page 42, that describes a array of categories to which the device belongs. One value MUST be EnOcean which is specified in DEVICE_CATEGORY.

Values for the additional service properties, DEVICE_DESCRIPTION, DEVICE_SERIAL as defined in *Device Service Registration* on page 42, are not specified here as no description nor application-level serial number are provided in the EnOcean standard protocol.

139.7.2 Import Situation

In import situations, the device's chip ID is uniquely set by the Base Driver, according to the one present in the teach-in message that originated the Device's creation. The service PID, see [7] *Persistent Identifier (PID)*, should also be generated and deterministically derived from the chip ID to allow reconstruction of a device without a new teach-in process after a framework restart.

139.7.3 Export Situation

In export situations:

1. The registering Client bundle sets the service PID of the EnOceanDevice object by itself, in a unique manner, and registers that object.
2. The chip ID (this device's EnOcean source ID when it issues messages) will be allocated by the Base Driver. The latter keeps a dictionary of the currently allocated chip IDs. The Client bundle must also set an ENOCEAN_EXPORT property in the registered device's Property Map.

The standard way to programmatically retrieve an exported chip ID from a given service PID is by using EnOceanHost's dedicated interface for this use.

The Base Driver MUST ensure the persistence of the CHIP_ID:SERVICE_PID mapping.

As an application developer, please refer to the documentation of your Base Driver to know its policies concerning exported chip ID updating, deletion and exhaustion.

139.7.4 Interface

The EnOceanDevice interface enables client bundles to:

- Get or set the security features of the device in a protected way;
- Retrieve the currently paired devices in the case of a receiver, as a collection of device IDs;
- Get the ID-based list of currently available RPCs for the device, as a Map of {manufacturerID:[functionId1, functionId2,...]};
- Invoke RPCs onto the device, through the invoke(EnOceanRPC,EnOceanHandler) call.

139.8 EnOcean Messages

EnOcean Messages are at the core of the EnOcean application layer as a whole and the EnOcean Equipment Profile specification, [2] *EnOcean Equipment Profiles v2.6.2*, in particular. Every exchange of information within EnOcean networks is done with a dedicated message. The EnOceanMessage interface provides a set of getters. The latter enables OSGi applications to get the information contained in the payload of an EnOcean message and defined as data and optional data of the EnOcean Serial Protocol Type 1 (RADIO) message (see Table 2 in Section "1.6.1 Packet description" of [4] *EnOcean System Specification - EnOcean Serial Protocol v1.17*).

This model enables reading both the EnOcean radio telegram data and the associated metadata that may be attached to it in a single object, EnOceanMessage.

In case the 'Optional Data' section gets missing at the lowest level (the radio access layer not following ESP protocol for instance) it is the responsibility of the Base Driver to mock the missing field's (dBm, destinationID, ...) values.

139.8.1 Mode of operation

Any EnOceanMessage object creation will be mirrored to Event Admin.

Details about the available topics, filters and properties can be found in *Event API* on page 771.

EnOceanMessage objects will be created only if the originating device has already been registered in the OSGi Service Registry, along with profile information.

139.8.2 Identification

The RORG of a message defines its shape and generic type; all the RORGs are defined in the EnOcean Radio Specification.

An addressed message will be encapsulated into an Addressed Telegram (ADT) by the base driver transparently; this means that from the application level, it will be represented under its original RORG, but with a valid destinationID.

A particular EnOcean Equipment Profile message is identified by three numbers: its RORG, and its FUNC, TYPE and EXTRA subtypes. In EnOcean, a (RORG, FUNC, TYPE) triplet is enough to identify a profile; though an EXTRA identifier is sometimes needed to identify a particular message layout for that profile.

Those identifiers allow for retrieving EnOceanMessageDescription objects within a registered EnOceanMessageDescriptionSet, which give the application more information to parse the message.

139.8.3 Interface

The methods available in the EnOceanMessage interface are:

- Identification methods, retrieving the message's profile, sender ID, optional destination ID, status;

- A method to get the raw bytes of payload data in the message. This data can then be passed to the deserializer of the EnOceanMessageDescription object to be converted to EnOceanChannel, which may -again- be documented (through EnOceanChannelDescription objects) or not.
- Link quality information read-only methods that mirror some of the 'Optional Data' header information.

139.9 EnOcean Message Description

EnOceanMessageDescription objects exposes only two methods:

- deserialize(byte[]): makes the user able to deserialize the payload bytes of a raw EnOceanMessage object, into a collection of EnOceanChannel objects.
- serialize(EnOceanChannel[]): serializes the input EnOceanChannel objects into a collection of bytes.

139.10 EnOcean Channel

The EnOceanChannel interface is the first step of an abstraction to generate or interpret EnOceanMessage channels with plain Java types.

The simple EnOceanChannel interface provides a way to separate the different fields in a message payload, knowing their offset and size in the byte array that constitutes the full message's payload.

At the EnOceanChannel level, the only way to get/set the information contained in the channel is through a pair of getRawValue() and setRawValue(byte[]) methods, which act on plain bytes.

Those bytes are meant right-aligned, and the number of those bytes is the size of the data field, floored up to the next multiple of 8. For instance, a 3-bit long channel would be encoded on one byte, all the necessary information starting from bit 0.

Every EnOceanMessage as described in the EEP Specification contains a various amount of channels, each of them being identified by their unique ID.

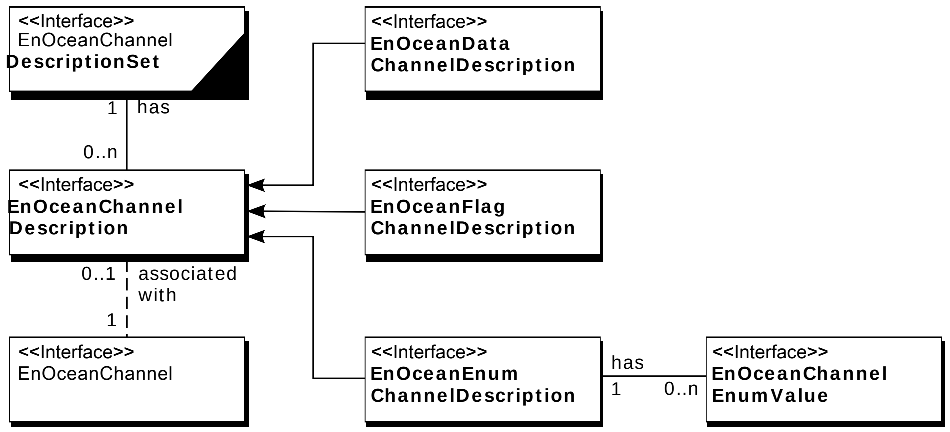
This ID, or channelID, is constituted of the “Shortcut” field of this channel from the EEP 2.5 Specification, [2] *EnOcean Equipment Profiles v2.6.2*, and a number fixed by the order of appearance of such a “Shortcut” in the specification.

This unique identifier links a Channel to an EnOceanChannelDescription object that provides more information to encode and decode that channel's information; see below for more details. This enables for loose coupling of the raw Channel itself and a richer, 3rd-party provided, information.

As an example, if the platform being developed is an electronic display that waits for Messages from a well-known temperature sensor, the Client bundle on the platform may interpret the Temperature Channels in every Temperature Message without needing an appropriate TemperatureChannelDescription object; it may directly cast and convert the Byte[] array of every received message to a properly valued Double and display that.

Otherwise, it could as well use the channelID to get a TemperatureChannelDescription object that would properly handle the deserialization process from the raw bytes to a proper, physical unit-augmented, result.

Figure 139.6 *EnOcean channel and EnOcean channel descriptions.*



139.11 EnOcean Channel Description

The EnOceanChannelDescription interface enables the description of all the various channels as specified in the EnOcean specification, as well as the description of channels issued by 3rd party actors.

Those description objects are retrieved from the registered EnOceanChannelDescriptionSet interface using an unique ID known as the channelID.

Here are the Channel types defined in this specification:

- TYPE_RAW: A collection of bytes. This type is used when the description is not provided, and is thus the default. For this type, the deserialize(byte[]) call actually returns a byte[] collection. The encryption key or a device ID on 4 bytes are examples of such raw types.
- TYPE_DATA: A scaled physical value. Used when the data can be mapped to a physical value; for instance, the 'WND – Wind Speed' channel is a raw binary value, in a range from 0 to 255, that will be mapped as a wind speed between 0 and 70 m/s. For this type, the deserialize(byte[]) call actually returns a Double value.
- TYPE_FLAG: A boolean value. Used when the Channel value can be either 1 or 0. The “Teach-In” Channel is a well-known example; this 1-bit field may either be 0 or 1, depending whether the Message is a teach-in one or not. For this type, the deserialize(byte[]) call actually returns a Boolean value.
- TYPE_ENUM: An enumeration of possible values. Used when the Channel can only take a discrete number of values. More complicated than TYPE_FLAG, enumerated types may have thresholds: for instance, the A5-30 “Digital Input- Input State (IPS)” channel is a 8-bit value which means “Contact closed” between 0 and 195, and “Contact open” from 196 to 255. For this type, the deserialize(byte[]) call actually returns an EnOceanChannelEnumValue object.

According to the channel type, the actual description object should implement one of the following specialized interfaces. This will ease the use of casting to the specialized interfaces on documented channels.

139.11.1 EnOcean Data Channel Description

The EnOceanDataChannelDescription interface inherits from EnOceanChannelDescription interface.

Two more methods give access to the integer input domain of the data channel (such as 0-255) and to the floating-point output range of it (such as -30.0°C – 24.5°C). A method is also present to retrieve the physical unit of the channel. The serialize(Object) and deserialize(byte[]) methods are implemented to easily convert from the raw byte[] collection to a Double, and vice versa.

Here are a few samples of such Channels:

Table 139.1 EnOcean Data Channel Description example

Short	Description	Possible implemented name	Domain	Range	Unit
TMP	Temperature	TemperatureScaledChannel_X	0..255	-10°..+30°	°C
HUM	Humidity	HumidityScaledChannel_X	0..250	0..100	%

139.11.2 EnOcean Flag Channel Description

The EnOceanFlagChannelDescription interface inherits from the EnOceanChannelDescription interface.

Those channels, are typically used for On/Off reporting values (like a switch); they have no additional methods, though the deserialize(byte[]) method converts the input bit into a proper Boolean object.

139.11.3 EnOcean Enumerated Channel Description

The EnOceanEnumChannelDescription interface inherits from the EnOceanChannelDescription interface.

The additional method provided to this interface is getPossibleValues(), which returns an array of the available EnOceanChannelEnumValue objects accessible to this channel. Every EnOceanChannelEnumValue object contains its integer input range and a String identifier that defines its meaning.

The serialize(Object) and deserialize(byte[]) methods of an EnOceanEnumChannelDescription object thus convert an integer input value (say, 156) to an EnOceanChannelEnumValue, and vice versa.

Here is an example that shows the input range and the associated EnOceanChannelEnumValue:

Table 139.2 EnOcean Enumerated Channel Description example

Device profile	EnOceanChannelEnumValue	Start	Stop	Meaning
Fan speed stage switch	FanStageSwitch_Stage3	0	144	Fan speed: Stage 3
	FanStageSwitch_Stage2	145	164	Fan speed: Stage 2
	FanStageSwitch_Stage1	165	189	Fan speed: Stage 1
	FanStageSwitch_Stage0	190	209	Fan speed: Stage 0

139.12 EnOcean Remote Management

Remote Management is a feature which allows EnOcean devices to be configured and maintained over the air using radio messages.

The Remote Procedure Calls, or RPCs - as defined by the EnOcean Remote Management specification, [3] *EnOcean System Specification - Remote Management v2.0* - are not related to any EnOcean Equipment Profile.

Note that EnOcean Remote Commissioning is detailed in an additional EnOcean document, [6] *EnOcean Remote Commissioning Summary v1.0.*

139.12.1 EnOcean RPC

An EnOceanRPC object enables client bundles to remotely manage EnOcean devices using already defined behavior.

RPCs are defined by a MANUFACTURER_ID (11 bits, 0x7FF for the EnOcean alliance) and a unique FUNCTION_ID code on 12 bits.

RPCs are called directly onto an EnOceanDevice object via the invoke(EnOceanRPC,EnOceanHandler) method, which accepts also a non-mandatory EnOceanHandler object as a parameter to retrieve the asynchronous answer.

Broadcasted RPCs can be addressed directly to the Base Driver using the relevant Event Admin topic; see *Event API* on page 771.

139.12.2 EnOcean Handler

Responses to RPCs are processed by the driver and sent back to a handler using notifyResponse(EnOceanRPC,byte[]) method when an EnOceanHandler is passed to the base driver.

139.13 Working With an EnOcean Device

139.13.1 Service Tracking

All discovered EnOcean devices in the local networks are registered under EnOceanDevice interface within the OSGi framework. Every time an EnOcean device appears or quits the network, the associated OSGi service is registered or unregistered in the OSGi service registry. Thanks to the EnOcean Base Driver, the OSGi service availability in the registry mirrors EnOcean device availability on EnOcean network, [1] *Pervasive Service Composition in the Home Network*.

Thanks to service events, a bundle is able to track the addition, modification and removal of an EnOceanDevice service.

The following example shows using a ServiceTracker to track EnOceanDevice services.

```
ServiceTracker<EnOceanDevice, EnOceanDevice> enOceanTracker =
    new ServiceTracker<>(bundleContext, EnOceanDevice.class, null);
enOceanTracker.open(); // open the tracker

...

// get a snaphot of the current EnOceanDevice services
EnOceanDevice[] enOceanDeviceSnapshot =
    enOceanTracker.getServices(new EnOceanDevice[0]);

...

enOceanTracker.close(); // close the tracker
```

139.14 Event API

EnOcean events must be delivered to the EventAdmin service by the EnOcean implementation, if present. EnOcean event topic follow the following form: org/osgi/service/enocean/EnOceanEvent/*SUBTOPIC*.

MESSAGE_RECEIVED and RPC_BROADCAST are the two available subtopics.

139.14.1 MESSAGE_RECEIVED

Properties (every event may dispatch some or all of the following properties):

- CHIP_ID – . The chip ID of the sending device.
- service.pid – The service PID of the exported device.
- RORG – The RORG (Radio Telegram Type) of the sending device.
- FUNC – The FUNC profile identifier of the sending device.
- TYPE – The TYPE profile identifier of the sending device.
- PROPERTY_MESSAGE – The EnOceanMessage object associated with this event.
- PROPERTY_EXPORTED – The presence of this property means that this message has actually been exported from a locally implemented EnOcean Device.

139.14.2 RPC_BROADCAST

This event is used whenever an RPC is broadcasted on EnOcean networks, in IMPORT or EXPORT situations.

Properties (every event may dispatch some or all of the following properties):

- MANUFACTURER_ID – The RPC's manufacturer ID.
- FUNCTION_ID – The RPC's function ID .
- PROPERTY_EXPORTED – The presence of this property means that this RPC has actually been exported from a locally implemented EnOcean Device.
- PROPERTY_RPC – The EnOceanRPC object associated with this event.

139.15 EnOcean Exceptions

The EnOceanException can be thrown and holds information about the different EnOcean layers. Here below, ESP stands for *EnOcean Serial Protocol*. The following errors are defined:

- ESP_UNEXPECTED_FAILURE – Operation was not successful.
- ESP_RET_NOT_SUPPORTED – The ESP command was not supported by the driver.
- ESP_RET_WRONG_PARAM – The ESP command was supplied wrong parameters.
- ESP_RET_OPERATION_DENIED – The ESP command was denied authorization.
- INVALID_TELEGRAM – The message was invalid.

139.16 Security

It is recommended that ServicePermission[EnOceanDevice|EnOceanHost, REGISTER|GET] be used sparingly and only for bundles that are trusted.

139.17 org.osgi.service.enocean

EnOcean Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.enocean; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.enocean; version="[1.0,1.1)"

139.17.1 Summary

- EnOceanChannel - Holds the raw value and channel identification info of an EnOceanChannel.
- EnOceanDevice - This interface represents a physical device that communicates over the EnOcean protocol.
- EnOceanEvent - Constants for use in EnOcean events.
- EnOceanException - This class contains code and definitions necessary to support common EnOcean exceptions.
- EnOceanHandler - The interface used to get callback answers from a RPC or a Message.
- EnOceanHost - This interface represents an EnOcean Host, a device that offers EnOcean networking features.
- EnOceanMessage - Holds the necessary methods to interact with an EnOcean message.
- EnOceanRPC - A very basic interface for RPCs.

139.17.2 public interface EnOceanChannel

Holds the raw value and channel identification info of an EnOceanChannel.

139.17.2.1 public String getChannelId()

Returns The unique ID of this channel.

139.17.2.2 public int getOffset()

Returns The offset, in bits, where this channel is found in the telegram.

139.17.2.3 public byte[] getRawValue()

□ Gets the raw value of this channel.

Returns corresponding value.

139.17.2.4 public int getSize()

Returns The size, in bits, of this channel.

139.17.2.5 public void setRawValue(byte[] rawValue)

rawValue

□ Sets the raw value of a channel.

139.17.3 public interface EnOceanDevice

This interface represents a physical device that communicates over the EnOcean protocol.

139.17.3.1 public static final String CHIP_ID = "enocean.device.chip_id"

Property name for the mandatory CHIP_ID of the device

139.17.3.2 public static final String DEVICE_CATEGORY = "EnOcean"

Property name for the mandatory DEVICE_CATEGORY of the device

139.17.3.3 **public static final String ENOCEAN_EXPORT = "enocean.device.export"**

Property name that defines if the device is exported or not. If present, the device is exported.

139.17.3.4 **public static final String FUNC = "enocean.device.profile.func"**

Property name for the radiotelegram functional type of the profile associated with this device.

139.17.3.5 **public static final String MANUFACTURER = "enocean.device.manufacturer"**

Property name for the manufacturer ID that may be specified by some teach-in messages.

139.17.3.6 **public static final String RORG = "enocean.device.profile.rorg"**

Property name for the radiotelegram main type of the profile associated with this device.

139.17.3.7 **public static final String SECURITY_LEVEL_FORMAT = "enocean.device.security_level_format"**

Property name for the security level mask for this device. The format of that mask is specified in EnOcean Security Draft.

139.17.3.8 **public static final String TYPE = "enocean.device.profile.type"**

Property name for the radiotelegram subtype of the profile associated with this device.

139.17.3.9 **public int getChipId()**

Returns The EnOcean device chip ID.

139.17.3.10 **public byte[] getEncryptionKey()**

□ Returns the current encryption key used by this device.

Returns The current encryption key, or null.

139.17.3.11 **public int getFunc()**

Returns The EnOcean profile FUNC, or -1 if unknown.

139.17.3.12 **public int[] getLearnedDevices()**

□ Gets the list of devices the device already has learned.

Returns The list of currently learned device's CHIP_IDs.

139.17.3.13 **public int getManufacturer()**

Returns The EnOcean manufacturer code, -1 if unknown.

139.17.3.14 **public int getRollingCode()**

□ Get the current rolling code of the device.

Returns The current rolling code in use with this device's communications.

139.17.3.15 **public int getRorg()**

Returns The EnOcean profile RORG.

139.17.3.16 **public Map<Integer, Integer> getRPCs()**

□ Retrieves the currently available RPCs to this device; those are stored using their manfufacturerId:commandId identifiers.

Returns A list of the available RPCs, in a `Map<Integer, Integer[]>` form.

139.17.3.17 public int getSecurityLevelFormat()

Returns The EnOcean security level format, or 0 as default (no security)

139.17.3.18 public int getType()

Returns The EnOcean profile TYPE, or -1 if unknown.

139.17.3.19 public void invoke(EnOceanRPC rpc, EnOceanHandler handler)

rpc

handler

□ Sends an RPC to the remote device.

Throws IllegalArgumentException–

139.17.3.20 public void remove()

□ Removes the device's OSGi service from OSGi service platform.

139.17.3.21 public void setEncryptionKey(byte[] key)

key the encryption key to be set.

□ Sets the encryption key of the device.

139.17.3.22 public void setFunc(int func)

func the EEP func of the device;

□ Manually sets the EEP FUNC of the device.

139.17.3.23 public void setLearningMode(boolean learnMode)

learnMode the desired state: true for learning mode, false to disable it.

□ Switches the device into learning mode.

139.17.3.24 public void setRollingCode(int rollingCode)

rollingCode the rolling code to be set or initiated.

□ Sets the rolling code of this device.

139.17.3.25 public void setType(int type)

type the EEP type of the device;

□ Manually sets the EEP TYPE of the device.

139.17.4 public final class EnOceanEvent

Constants for use in EnOcean events.

139.17.4.1 public static final String PROPERTY_EXPORTED = "enocean.message.is_exported"

Property key used to tell apart messages that are exported or imported.

139.17.4.2 public static final String PROPERTY_MESSAGE = "enocean.message"

Property key for the EnOceanMessage object embedded in an event.

139.17.4.3 public static final String PROPERTY_RPC = "enocean.rpc"

Property key for the EnOceanRPC object embedded in an event.

139.17.4.4 public static final String TOPIC_MSG_RECEIVED = "org/osgi/service/enocean/EnOceanEvent/MESSAGE_RECEIVED"

Main topic for all OSGi dispatched EnOcean messages, imported or exported.

139.17.4.5 public static final String TOPIC_RPC_BROADCAST = "org/osgi/service/enocean/EnOceanEvent/RPC_BROADCAST"

Main topic for all OSGi broadcast EnOcean RPCs, imported or exported.

139.17.5 public class EnOceanException extends Exception

This class contains code and definitions necessary to support common EnOcean exceptions. This class is mostly used with low-level, gateway-interacting code : EnOceanHost.

139.17.5.1 public static final short ESP_RET_NOT_SUPPORTED = 2

Operation is not supported by the target device.

139.17.5.2 public static final short ESP_RET_OPERATION_DENIED = 4

The operation was denied.

139.17.5.3 public static final short ESP_RET_WRONG_PARAM = 3

One of the parameters was badly specified or missing.

139.17.5.4 public static final short ESP_UNEXPECTED_FAILURE = 1

Unexpected failure.

139.17.5.5 public static final short INVALID_TELEGRAM = 240

The message was invalid.

139.17.5.6 public static final short SUCCESS = 0

SUCCESS status code.

139.17.5.7 public EnOceanException(String errordesc)

errordesc exception error description

□ Constructor for EnOceanException

139.17.5.8 public EnOceanException(int errorCode, String errorDesc)

errorCode the error code.

errorDesc the description.

□ Constructor for EnOceanException

139.17.5.9 public EnOceanException(int errorCode)

errorCode An error code.

□ Constructor for EnOceanException

139.17.5.10 public int errorCode()

□ Constructor for EnOceanException

Returns An EnOcean error code, defined by the EnOcean Forum working committee or an EnOcean vendor.

139.17.6 public interface EnOceanHandler

The interface used to get callback answers from a RPC or a Message.

139.17.6.1 public void notifyResponse(EnOceanRPC original, byte[] payload)

original the original EnOceanRPC that originated this answer.

payload the payload of the response; may be deserialized to an EnOceanRPC object.

□ Notifies of the answer to a RPC.

139.17.7 public interface EnOceanHost

This interface represents an EnOcean Host, a device that offers EnOcean networking features.

139.17.7.1 public static final Object HOST_ID

The unique ID for this Host: this matches the CHIP_ID of the EnOcean Gateway Chip it embodies.

139.17.7.2 public static final int REPEATER_LEVEL_OFF = 0

repeater level to disable repeating; this is the default.

139.17.7.3 public static final int REPEATER_LEVEL_ONE = 1

repeater level to repeat every telegram at most once.

139.17.7.4 public static final int REPEATER_LEVEL_TWO = 2

repeater level to repeat every telegram at most twice.

139.17.7.5 public String apiVersion() throws EnOceanException

□ Returns the chip's API version info (cf. ESP3 command 0x03: CO_RD_VERSION)

Returns a String object containing the API version info.

Throws `EnOceanException`– if any problem occurs.

139.17.7.6 public String appVersion() throws EnOceanException

□ Returns the chip's application version info (cf. ESP3 command 0x03: CO_RD_VERSION)

Returns a String object containing the application version info.

Throws `EnOceanException`– if any problem occurs.

139.17.7.7 public int getBaseID() throws EnOceanException

□ Gets the BASE_ID of the chip, if set (cf. ESP3 command 0x08: CO_RD_IDBASE)

Returns the BASE_ID of the device as defined in EnOcean specification

Throws `EnOceanException`– if any problem occurs.

139.17.7.8 public int getChipId(String servicePID) throws EnOceanException

servicePID

□ Retrieves the CHIP_ID associated with the given servicePID, if existing on this chip.

Returns the associated CHIP_ID of the exported device.

Throws `EnOceanException`– if any problem occurs.

139.17.7.9 public int getRepeaterLevel() throws EnOceanException

□ Gets the current repeater level of the host (cf. ESP3 command 0x0A: CO_RD_REPEATER)

Returns one of the Repeater Level constants as defined above.

Throws `EnOceanException` – if any problem occurs.

139.17.7.10 public void reset() throws EnOceanException

□ Reset the EnOcean Host (cf. ESP3 command 0x02: CO_WR_RESET)

Throws `EnOceanException` – if any problem occurs.

139.17.7.11 public void setBaseID(int baseID) throws EnOceanException

baseID to be set.

□ Sets the base ID of the device, may be used up to 10 times (cf. ESP3 command 0x07: CO_WR_IDBASE)

Throws `EnOceanException` – if any problem occurs.

139.17.7.12 public void setRepeaterLevel(int level) throws EnOceanException

level one of the Repeater Level constants as defined above.

□ Sets the repeater level on the host (cf. ESP3 command 0x09: CO_WR_REPEATER)

Throws `EnOceanException` – if any problem occurs.

139.17.8 public interface EnOceanMessage

Holds the necessary methods to interact with an EnOcean message.

139.17.8.1 public byte[] getBytes()

□ Gets the bytes corresponding to the whole message, including the CRC. The generated byte[] array may be sent to an EnOcean gateway and is conform to EnOcean Radio Protocol.

Returns The serialized byte list corresponding to the binary message.

139.17.8.2 public int getDbm()

□ Returns the average RSSI on all the received subtelegrams, including redundant ones.

Returns The average RSSI perceived.

139.17.8.3 public int getDestinationId()

Returns the message's destination ID, or -1

139.17.8.4 public int getFunc()

Returns the message's FUNC

139.17.8.5 public byte[] getPayloadBytes()

□ Returns the payload bytes of this message.

Returns corresponding value.

139.17.8.6 public int getRorg()

Returns the message's RORG

139.17.8.7 public int getSecurityLevelFormat()

□ Returns the security level of this message, as specified in the 'Security of EnOcean Radio Networks' draft, section 4.2.1.3.

Returns The security level format.

139.17.8.8 public int getSenderId()

Returns the message's Sender ID

139.17.8.9 public int getStatus()

□ Gets the current EnOcean status of the Message. The 'status' byte is actually a bitfield that mainly holds repeater information, teach-in status, and more or less information depending on the radiotelegram type.

Returns the current EnOcean status of this message.

139.17.8.10 public int getSubTelNum()

□ Returns the number of subtelegrams (usually 1) this Message carries.

Returns The number of subtelegrams in the case of multiframe messages.

139.17.8.11 public int getType()

Returns the message's TYPE

139.17.9 public interface EnOceanRPC

A very basic interface for RPCs.

139.17.9.1 public static final String FUNCTION_ID = "enocean.rpc.function_id"

The Function ID property string, used in EventAdmin RPC broadcasting.

139.17.9.2 public static final String MANUFACTURER_ID = "enocean.rpc.manufacturer_id"

The Manufacturer ID property string, used in EventAdmin RPC broadcasting.

139.17.9.3 public int getFunctionId()

□ Gets the functionID for this RPC.

Returns function id.

139.17.9.4 public int getManufacturerId()

□ Gets the manufacturerID for this RPC.

Returns manufacturer id.

139.17.9.5 public String getName()

□ Get a friendly name for the RPC

Returns the name.

139.17.9.6 public byte[] getPayload()

□ Gets the current payload of the RPC.

Returns the payload, in bytes, of this RPC.

139.17.9.7 public int getSenderId()

□ Sets the RPC's senderID. This member has to belong to EnOceanRPC interface, for the object may be sent as a standalone using EventAdmin for instance.

Returns sender id.

139.17.9.8 public void setSenderId(int chipId)

chipId

□ Sets the RPC's senderID.

139.18 org.osgi.service.enocean.descriptions

EnOcean Descriptions Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.enocean.descriptions; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.enocean.descriptions; version="[1.0,1.1)"`

139.18.1 Summary

- `EnOceanChannelDescription` - Public and registered description interface for a channel.
- `EnOceanChannelDescriptionSet` - This interface represents an EnOcean Channel Description Set.
- `EnOceanChannelEnumValue` - This transitional interface is used to define all the possible values taken by an enumerated channel.
- `EnOceanDataChannelDescription` - Subinterface of EnOceanChannelDescription that describes physical measuring channels.
- `EnOceanEnumChannelDescription` - Subinterface of EnOceanChannelDescription that describes enumerated channels.
- `EnOceanFlagChannelDescription` - Subinterface of EnOceanChannelDescription that describes boolean channels.
- `EnOceanMessageDescription` - This interface represents an EnOcean Message Description.
- `EnOceanMessageDescriptionSet` - This interface represents an EnOcean Message Description Set.

139.18.2 public interface EnOceanChannelDescription

Public and registered description interface for a channel. Encompasses all the possible subtypes for a channel.

139.18.2.1 public static final String CHANNEL_ID = "enocean.channel.description.channel_id"

The unique ID of this EnOceanChannelDescription object.

139.18.2.2 public static final String TYPE_DATA = "enocean.channel.description.data"

A DATA channel maps itself to a Double value representing a physical measure.

139.18.2.3 public static final String TYPE_ENUM = "enocean.channel.description.enum"

An ENUM channel maps itself to one between a list of discrete EnOceanChannelEnumValue "value objects".

139.18.2.4 public static final String TYPE_FLAG = "enocean.channel.description.flag"

A FLAG channel maps itself to a Boolean value.

139.18.2.5 public static final String TYPE_RAW = "enocean.channel.description.raw"

A RAW channel is only made of bytes.

139.18.2.6 public Object deserialize(byte[] bytes)

bytes the right-aligned raw bytes.

□ Tries to deserialize a series of bytes into a documented value object (raw bytes, Double or EnOceanChannelEnumValue. Of course this method will be specialized for each EnOceanChannelDescription subinterface, depending on the type of this channel.

Returns a value object.

Throws IllegalArgumentException–

139.18.2.7 public String getType()

□ Retrieves the type of the channel.

Returns one of the above-described types.

139.18.2.8 public byte[] serialize(Object obj)

obj the value of the channel.

□ Tries to serialize the channel into a series of bytes.

Returns the right-aligned value, in raw bytes, of the channel.

Throws IllegalArgumentException–

139.18.3 public interface EnOceanChannelDescriptionSet

This interface represents an EnOcean Channel Description Set. EnOceanChannelDescriptionSet is registered as an OSGi Service. Provides a method to retrieve the EnOceanChannelDescription objects it documents.

139.18.3.1 public EnOceanChannelDescription getChannelDescription(String channelId)

channelId the unique string identifier of the description object.

□ Retrieves a EnOceanChannelDescription object according to its identifier.

Returns The corresponding EnOceanChannelDescription object, or null.

Throws IllegalArgumentException– if the supplied String is invalid, null, or other reason.

139.18.4 public interface EnOceanChannelEnumValue

This transitional interface is used to define all the possible values taken by an enumerated channel.

139.18.4.1 public String getDescription()

□ A non-mandatory description of what this enumerated value is about.

Returns the description of this channel.

139.18.4.2 public int getStart()

□ The start value of the enumeration.

Returns the start value.

139.18.4.3 public int getStop()

□ The stop value of the enumeration.

Returns the stop value.

139.18.5 public interface EnOceanDataChannelDescription extends EnOceanChannelDescription

Subinterface of EnOceanChannelDescription that describes physical measuring channels.

139.18.5.1 public int getDomainStart()

□ The start of the raw input range for this channel.

Returns the domain start.

139.18.5.2 public int getDomainStop()

□ The end of the raw input range for this channel.

Returns the domain stop.

139.18.5.3 public double getRangeStart()

□ The scale start at which this channel will be mapped to (-20,0°C for instance)

Returns the range start.

139.18.5.4 public double getRangeStop()

□ The scale stop at which this channel will be mapped to (+30,0°C for instance)

Returns the range stop.

139.18.5.5 public String getUnit()

□ The non-mandatory physical unit description of this channel.

Returns the unit as a String

139.18.6 public interface EnOceanEnumChannelDescription extends EnOceanChannelDescription

Subinterface of EnOceanChannelDescription that describes enumerated channels.

139.18.6.1 public EnOceanChannelEnumValue[] getPossibleValues()

□ Gets all the possible value for this channel.

Returns corresponding value(s).

139.18.7 public interface EnOceanFlagChannelDescription extends EnOceanChannelDescription

Subinterface of EnOceanChannelDescription that describes boolean channels.

139.18.8 public interface EnOceanMessageDescription

This interface represents an EnOcean Message Description.

139.18.8.1 public EnOceanChannel[] deserialize(byte[] bytes)

bytes to be deserialized.

□ Deserializes an array of bytes into the EnOceanChannels available to the payload, if possible.

Returns deserialized value.

Throws `IllegalArgumentException`– if the actual instance type of the message is not compatible with the bytes it is fed with (RORG to begin with).

139.18.8.2 public String getMessageDescription()

Returns the message description containing the RORG, (and the FUNC, and the TYPE if available), as well as, the EEP's "title" (e.g. for F60201: Rocker Switch, 2 Rocker; Light and Blind Control - Application Style 1).

139.18.8.3 public byte[] serialize(EnOceanChannel[] channels)

channels to be serialized.

□ Serializes a series of EnOceanChannel objects into the corresponding byte[] sequence.

Returns serialized value.

Throws `IllegalArgumentException` – if the given channels is null.

139.18.9 public interface EnOceanMessageDescriptionSet

This interface represents an EnOcean Message Description Set. EnOceanMessageDescriptionSet is registered as an OSGi Service. Provides method to retrieve the EnOceanMessageDescription objects it documents.

139.18.9.1 public EnOceanMessageDescription getMessageDescription(int rorg, int func, int type, int extra)

rorg the radio telegram type of the message.

func The func subtype of this message.

type The type subselector.

extra Some extra information; some EnOceanMessageDescription objects need an additional specifier. If not needed, has to be set to -1.

□ Retrieves a EnOceanMessageDescription object according to its identifiers. See EnOcean Equipment Profile Specification for more details.

Returns The EnOceanMessageDescription object looked for, or null.

Throws `IllegalArgumentException` – if there was an error related to the input arguments.

139.19 References

[1] *Pervasive Service Composition in the Home Network*
Bottaro, A., Gérodolle, A., Lalanda, P., 21st IEEE International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 2007

[2] *EnOcean Equipment Profiles v2.6.2*
EnOcean Alliance, https://www.enocean-alliance.org/en/enocean_standard/, November 19, 2014

[3] *EnOcean System Specification - Remote Management v2.0*
EnOcean Alliance, March 06, 2014

[4] *EnOcean System Specification - EnOcean Serial Protocol v1.17*
EnOcean Alliance, August 2, 2011

[5] *EnOcean System Specification - Security of EnOcean Radio Networks v1.9*
EnOcean Alliance, July 26, 2013

[6] *EnOcean Remote Commissioning Summary v1.0*
EnOcean Alliance, https://www.enocean-alliance.org/en/downloads/, December 01, 2014

[7] *Persistent Identifier (PID)*
OSGi Core Release, Service Layer

140 Whiteboard Specification for Jakarta™ Servlet

Version 2.0

140.1 Introduction

Servlets have become a popular and widely supported mechanism for providing dynamic content on the Internet. While servlets are defined in the [4] *Jakarta Servlet 5.0 Specification*, the Servlet Whiteboard Specification provides a light and convenient way of using servlets, servlet filters, servlet listeners and web resources in an OSGi environment through the use of the [7] *Whiteboard Pattern*.

The Servlet Whiteboard Specification supports:

- *Registering Servlets* - Registering a servlet in the Service Registry makes it available to be bound to an endpoint to serve content over the network.
- *Registering Servlet Filters* - Servlet filters support pre- and post-processing of servlet requests and responses. Servlet filters can be registered in the Service Registry to include them in the handling pipeline.
- *Registering Resources* - Resources such as HTML files, JavaScript, image files, and other static resources can be made available over the network by registering resource services.
- *Registering Servlet Listeners* - The servlet specification defines a variety of listeners, which receive callbacks when certain events take place.

Implementations of this specification can support the following versions of the HTTP protocol:

- [1] *HTTP 1.0 Specification RFC-1945*
- [2] *HTTP 1.1 Specifications RFCs 7230-7235*
- [3] *HTTP/2 Specifications*

Alternatively, implementations of this service can support other protocols if these protocols can conform to the semantics of the Jakarta Servlet API.

Implementations of this specification must support version 5.0 or later of the Jakarta Servlet API.

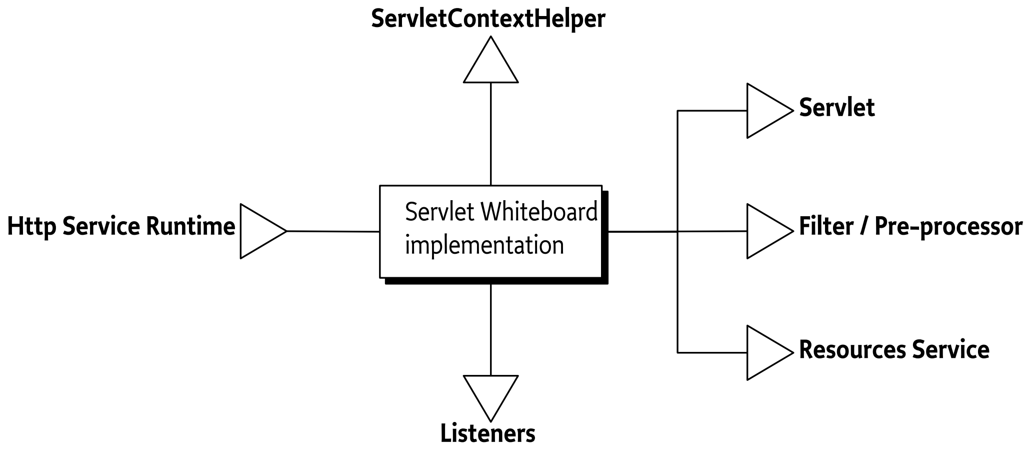
140.1.1 Entities

This specification defines the following entities:

- *Servlet Whiteboard service* - An object registered in the Service Registry under one of the Whiteboard service interfaces defined by this specification.
- *Servlet Whiteboard implementation* - An implementation that processes Servlet Whiteboard services.
- *Http Service Runtime service* - Service providing runtime introspection into the Servlet Whiteboard implementation.
- *Listener* - Various listeners can be registered to receive notifications about servlet or Http Session events.

- *Resource Service* - A service thats binds static resources.
- *Servlet* - Component that dynamically generates web pages or other resources provided over the network.
- *Servlet Context Helper* - A service to control the behavior of the Servlet Context.
- *Servlet Filter* - Can be used to augment or transform web resources or for cross-cutting functionality such as security, common widgets or otherwise.

Figure 140.1 *Servlet Whiteboard Overview Diagram*



140.2 The Servlet Context

The servlet specification defines the ServletContext which is provided to servlets at runtime by the container. Whiteboard services defined by this specification are also provided with a ServletContext. The behavior of this Servlet Context can be influenced by providing a ServletContextHelper service. A custom ServletContextHelper can provide resources, mime-types, handle security and supports a number of methods from the ServletContext.

The Servlet Whiteboard implementation must create a separate ServletContext instance for each ServletContextHelper service. Whiteboard services can be associated with the Servlet Context Helper by using the osgi.http.whiteboard.context.select property. If this property is not set, the *default* Servlet Context Helper is used.

To achieve the required behavior for ServletContext.getClassLoader each bundle must be provided with a separate Servlet Context instance to serve the class loader of the Whiteboard services for that bundle. For more information see getClassLoader in Table 140.2 on page 789.

Some implementations of the ServletContextHelper may be implemented using a Service Factory, for example to provide resources from the associated bundle, as the *default* implementation does. Therefore the Whiteboard implementation must get the Servlet Context Helper using the Bundle Context of the bundle that registered the Whiteboard service.

Some environments may use [8] *Core Service Hooks* to isolate ServletContextHelper service registrations. The Whiteboard implementation must check that the bundle registering the Whiteboard service has the ability to find the ServletContextHelper service before allowing the Whiteboard service to bind to the Servlet Context Helper. This can be done by calling one of the getServiceReferences methods on the Bundle Context of bundle that registered the Whiteboard service.

Table 140.1 *Service registration properties for* *ServletContextHelper* *services.*

Service Property	Type	Description
osgi.http.whiteboard.context.name name	String *required*	Name of the Servlet Context Helper. This name can be referred to by Whiteboard services via the osgi.http.whiteboard.context.select property. The syntax of the name is the same as the syntax for a Bundle Symbolic Name. The default Servlet Context Helper is named default. To override the default, register a custom ServletContextHelper service with the name default. If multiple Servlet Context Helper services are registered with the same name, the one that is first in ranking order, as specified in ServiceReference.compareTo, is used. Registrations with an invalid or unspecified name are not used and reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_NAME.
osgi.http.whiteboard.context.path path	String *required*	Additional prefix to the context path for servlets. This property is mandatory. Valid characters are specified in IETF RFC 3986, section 3.3. The context path of the default Servlet Context Helper is /. A custom default Servlet Context Helper may use an alternative path. If the path is invalid or unspecified, the service is not used and reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_PATH.
context.init.*	String *optional*	Properties starting with this prefix are provided as init parameters through the ServletContext.getInitParameter and ServletContext.getInitParameterNames methods. The context.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_CONTEXT_INIT_PARAM_PREFIX.

Multiple ServletContextHelper services can have identical or overlapping osgi.http.whiteboard.context.path values. A matching servlet or resource is located as follows:

1. The Servlet Context Helper service with the longest matching path is matched first.
2. In the case of multiple Servlet Context Helper services with the same path, the services are searched in ranking order, as specified in ServiceReference.compareTo, for a match.

For example, if two ServletContextHelper services are registered as follows

```
osgi.http.whiteboard.context.path = /foo
osgi.http.whiteboard.context.path = /foo/bar
```

Then a request for http://localhost/foo/bar/someServlet is looked up in the following order:

1. /foo/bar context looking for a pattern to match /someServlet
2. /foo context looking for a pattern to match /bar/someServlet

Note that whole path segments must match. Therefore the following request can only be handled by the Servlet Context Helper registered under the /foo path: http://localhost/foo/bars/someOtherServlet.

For details on the association process between servlet, servlet filter, resource and listener services and the ServletContextHelper see *Common Whiteboard Properties* on page 791.

If a Servlet Context Helper can not be used, for example because it is shadowed by another Servlet Context Helper service with the same name but with a higher ranking, this is reflected in the FailedServletContextDTO. Similarly, if an alternative default Servlet Context Helper is provided, the default Servlet Context Helper provided by the Servlet Whiteboard implementation is not used and represented in a failure DTO.

An example Servlet Context Helper defined using Declarative Services annotations can be found below, it prefixes the path with /myapp for any associated whiteboard service. Additionally, it serves static resources from a non-standard location, a content delivery network. Other methods use the default ServletContextHelper implementation.

```
@Component(service = ServletContextHelper.class, scope = ServiceScope.BUNDLE)
@HttpWhiteboardContext(name = "my-context", path = "/myapp")
public class CDNServletContextHelper extends ServletContextHelper {
    public URL getResource(String name) {
        try {
            return new URL("http://acmecdn.com/myapp/" + name);
        } catch (MalformedURLException e) {
            return null;
        }
    }
}
```

The following sections outline the methods a custom ServletContextHelper can override and the behavior of the *default* implementation.

140.2.1 String getMimeType(String)

Called to provide the MIME type for a resource.

Default Behavior - Always returns null.

140.2.2 String getRealPath(String)

Called to support the ServletContext.getRealPath method.

Default Behavior - Always returns null.

140.2.3 URL getResource(String)

Obtain a URL for a given resource request.

Default Behavior - Assumes the resources are in the bundle registering the Whiteboard service. Its Bundle.getEntry method is called to obtain a URL to the resource. The default Servlet Context Helper implementation assumes the path to be relative to the bundle's root.

140.2.4 Set<String> getResourcePaths(String)

Called to support the ServletContext.getResourcePaths method. Returns all the matching resources for the path.

Default Behavior - Assumes the resources are in the bundle registering the Whiteboard service. Its Bundle.findEntries method is called to obtain the listing.

140.2.5 Security Handling

The handleSecurity method is invoked to handle implementation-defined security on the request. It is invoked before the request is sent to the filter-servlet pipeline.

When the request returns from the filter-servlet pipeline the finishSecurity method is called. This method can be used by the security handling mechanism to clean up any context associated with the current request. finishSecurity is only called if handleSecurity returned true for the specified request. If an exception occurs during processing of the pipeline, finishSecurity is still called. This allows to clean up regardless of the result of the pipeline.

In the case a request is dispatched either using the include or forward method handleSecurity and finishSecurity are called again on this new context. These calls are nested within the originating request. Servlet Context Helpers that implement these methods must be prepared to deal with such nested invocations.

Default Behavior - handleSecurity always returns true. finishSecurity does nothing by default.

140.2.6 Behavior of the Servlet Context

The ServletContext provided to Whiteboard services is based on the associated ServletContextHelper, Whiteboard service registration properties and the underlying servlet container.

Methods to programmatically add servlets, servlet filters and listeners are not supported on the ServletContext. Such functionality is available by registering these entities as Whiteboard services.

Table 140.2 Behavior of ServletContext methods.

ServletContext method	Since	Description
addFilter(...)	3.0	Throws UnsupportedOperationException.
addListener(...)	3.0	Throws UnsupportedOperationException.
addServlet(...)	3.0	Throws UnsupportedOperationException.
createFilter(Class)	3.0	Throws UnsupportedOperationException.
createListener(Class)	3.0	Throws UnsupportedOperationException.
createServlet(Class)	3.0	Throws UnsupportedOperationException.
declareRoles(String ...)	3.0	Throws UnsupportedOperationException.
getAttribute(String)	2.0	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
getAttributeNames()	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
getClassLoader()	3.0	Returns the class loader of the bundle that registered the Whiteboard service. An implementation of this specification can achieve this by returning separate façades of the ServletContext to each Whiteboard service. Each façade accesses the Whiteboard service's Bundle Wiring to obtain its class loader.
getContext(String)	2.1	Backed by the Servlet Container.
getContextPath()	2.5	Return the web context path of the Servlet Context. This takes into account the osgi.http.whiteboard.context.path of the Servlet Context Helper and the path of the Http runtime.
getDefaultSessionTrackingModes()	3.0	Backed by the Servlet Container.
getEffectiveMajorVersion()	3.0	Backed by the Servlet Container.
getEffectiveMinorVersion()	3.0	Backed by the Servlet Container.
getEffectiveSessionTrackingModes()	3.0	Backed by the Servlet Container.
getFilterRegistration(String)	3.0	Backed by the Servlet Container.
getFilterRegistrations()	3.0	Backed by the Servlet Container.
getInitParameter(String)	2.2	From context.init.* service registration properties.
getInitParameterNames()	2.2	From context.init.* service registration properties.
getJspConfigDescriptor()	3.0	Returns null.

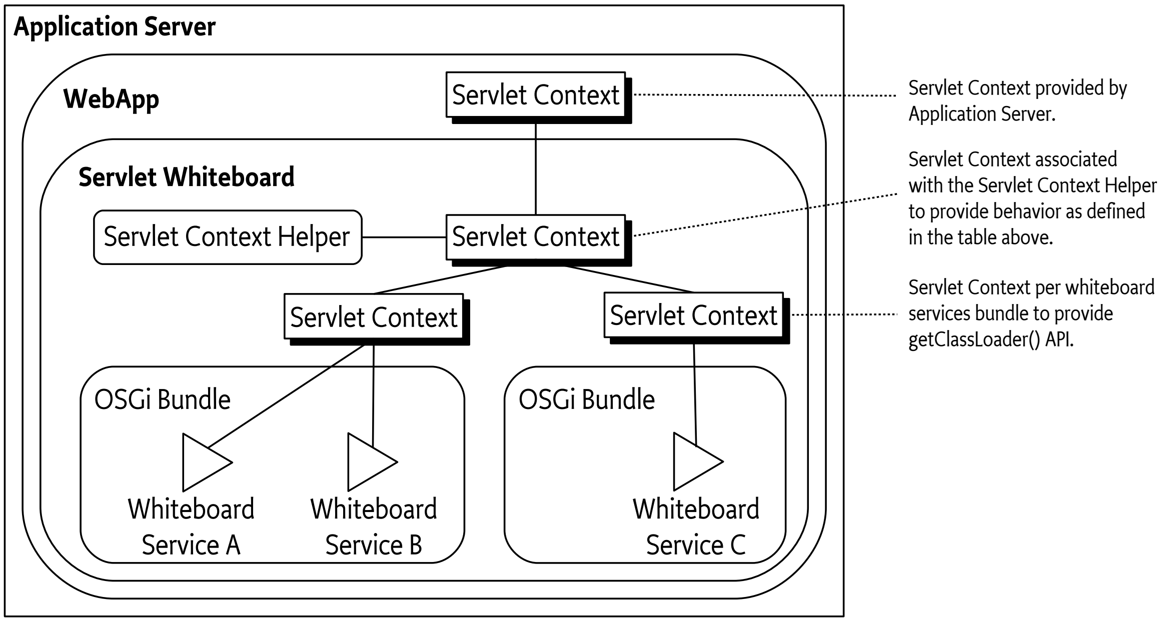
ServletContext method	Since	Description
getMajorVersion()	2.1	Backed by the Servlet Container.
getMimeType(String)	2.1	Backed by the ServletContextHelper.
getMinorVersion()	2.1	Backed by the Servlet Container.
getNamedDispatcher(String)	2.2	Provides the Whiteboard servlet with the specified name, provided through the osgi.http.whiteboard.servlet.name property, if associated with this Servlet Context Helper. If multiple servlets have the same name and are associated with this Servlet Context Helper, then the servlet that is first in ranking order, as specified in ServiceReference.compareTo, is used.
getRealPath(String)	2.0	Backed by the ServletContextHelper.
getResource(String)	2.1	Backed by the ServletContextHelper.
getRequestDispatcher(String)	2.1	If the argument matches a servlet associated with this Servlet Context Helper, this will be returned.
getResourceAsStream(String)	2.1	Backed by the ServletContextHelper.
getResourcePaths(String)	2.3	Backed by the ServletContextHelper.
getServlet(String)	2.0	Deprecated. Backed by the Servlet Container.
getServletContextName()	2.2	The name of the ServletContextHelper provided via the osgi.http.whiteboard.context.name service property.
getServletNames()	2.0	Deprecated. Backed by the Servlet Container.
getServletRegistration(String)	3.0	Backed by the Servlet Container.
getServletRegistrations()	3.0	Backed by the Servlet Container.
getServlets()	2.0	Deprecated. Backed by the Servlet Container.
getServerInfo()	2.0	Backed by the Servlet Container.
getSessionCookieConfig()	3.0	Returns a SessionCookieConfig object. This object is read-only and all setters throw a IllegalStateException.
getVirtualServerName()	3.1	Backed by the Servlet Container.
log(String)	2.0	Backed by the Servlet Container.
log(Exception, String)	2.0	Deprecated. Backed by the Servlet Container.
log(String, Throwable)	2.1	Backed by the Servlet Container.
removeAttribute(String)	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
setAttribute(String, Object)	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
setInitParameter(String, String)	3.0	Throws IllegalStateException. The ServletContext has already been initialized.
setSessionTrackingModes(Set)	3.0	Throws IllegalStateException. The ServletContext has already been initialized.

140.2.7 Relation to the Servlet Container

Implementations of this specification will often be backed by existing servlet containers or a Jakarta EE application server. There may also exist implementations which bridge into a servlet container into which the OSGi Framework has been deployed as a Web Application.

In bridged situations the Servlet Whiteboard implementation will live in one servlet context and all Whiteboard services registered by this implementation will be backed by the same underlying Servlet Context. However, to exhibit the behavior described in Table 140.2 on page 789 different Servlet Context objects may be required. Therefore an implementation of this specification may need to create additional ServletContext objects which delegate certain functionality to the ServletContextHelper and other functionality to the Servlet Context of the Web Application, yet further functionality can be obtained otherwise. In such cases the relationship may look like the below figure.

Figure 140.2 Servlet Context entities and their relation


Application Server
WebApp
Servlet Context
Servlet Context provided by Application Server.
Servlet Whiteboard
Servlet Context Helper
Servlet Context
Servlet Context associated with the Servlet Context Helper to provide behavior as defined in the table above.
Servlet Context
Servlet Context
Servlet Context per whiteboard services bundle to provide getClassLoader() API.
OSGi Bundle
Whiteboard Service A
Whiteboard Service B
OSGi Bundle
Whiteboard Service C

Where Table 140.2 on page 789 states *Backed by the Servlet Container* and the Servlet Whiteboard implementation is deployed in bridged mode, the API call can be forwarded to the top-level Servlet Context. If the Servlet Whiteboard implementation is not deployed in bridged mode, it must provide another means to handle these APIs.

In bridged deployments, the implementation needs to ensure the following:

1. That Whiteboard services are provided with the correct ServletContext keeping in mind that each distinct ServletContextHelper should be associated with a separate ServletContext object, which in turn may delegate certain requests to the underlying shared ServletContext as described in the table above.
2. That Http Sessions are not shared amongst servlets registered with different ServletContextHelpers. That is, HttpRequest.getSession calls must provide different sessions per associated ServletContextHelper. Http Sessions are defined in chapter 7 of the [4] *Jakarta Servlet 5.0 Specification.*

140.3 Common Whiteboard Properties

Whiteboard servlet, servlet filter, resource and listener services support common service registration properties to associate them with a ServletContextHelper and/or a Servlet Whiteboard implementation.

Table 140.3 Common properties

Service Property	Type	Description

osgi.http.whiteboard.context.select
HttpWhiteboardContextSelect
	String
	optional
		An LDAP-style filter to select the associated ServletContextHelper service to use. Any service property of the Servlet Context Helper can be filtered on. If this property is missing the default Servlet Context Helper is used.
		For example, to select a Servlet Context Helper with name myCTX provide the following value:
		`(osgi.http.whiteboard.context.name=myCTX)`
		To select all Servlet Context Helpers provide the following value:
		`(osgi.http.whiteboard.context.name=*)`
		If no matching context exists this is reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_SELECT.

osgi.http.whiteboard.target HttpWhiteboardTarget	String *optional*	The value of this service property is an LDAP-style filter expression to select the Servlet Whiteboard implementation(s) to handle this Whiteboard service. The LDAP filter is used to match HttpServiceRuntime services. Each Servlet Whiteboard *implementation* exposes exactly one HttpServiceRuntime service. This property is used to associate the Whiteboard service with the Servlet Whiteboard implementation that registered the HttpServiceRuntime service. If this property is not specified, all Servlet Whiteboard implementations can handle the service. See HTTP_WHITEBOARD_TARGET.

If multiple Servlet Context Helper services match the osgi.http.whiteboard.context.select property the servlet, filter, resource or listener will be registered with all these Servlet Context Helpers. To avoid multiple init and destroy calls on the same instance, servlets and filters should be registered as Prototype Service Factory.

140.4 Registering Servlets

Servlets can be registered with the Servlet Whiteboard implementation by registering them as Whiteboard services. This means that Servlet implementations are registered in the Service Registry under the jakarta.servlet.Servlet interface.

Servlets are registered with one or more pattern through the osgi.http.whiteboard.servlet.pattern service property. Each pattern defines the URL context that will trigger the servlet to handle the request. They are defined by the [4] *Jakarta Servlet 5.0 Specification* in section 12.2, *Specification of Mappings*. The mapping rules are:

- A string beginning with a '/' character and ending with a "/*" suffix is used for path mapping.
- A string beginning with a "*." prefix is used as an extension mapping.
- The empty string ("") is a special URL pattern that exactly maps to the application's context root. That is, requests of the form http://host:port/<context-root>/. In this case the path info is "/" and the servlet path and context path are the empty string ("").
- A string containing only the '/' character indicates the "default" servlet of the application. In this case, the servlet path is the request URI minus the context path and the path info is null.

- All other strings are used for exact matches only.

Servlet and resource service registrations associated with a single Servlet Context share the same namespace. In case of identical registration patterns, the service that is first in ranking order, as specified in ServiceReference.compareTo, is selected as the service handling a request. That is, Whiteboard servlets that have patterns shadowed by other Whiteboard services associated with the *same* Servlet Context are represented in the failure DTOs.

The above rules can cause servlets that are already bound becoming unbound if a better match arrives. This ensures a predictable end result regardless of the order in which services are registered.

A servlet may be registered with the property osgi.http.whiteboard.servlet.name which can be used by servlet filters to address this servlet. If the servlet service does not have this property, the servlet name defaults to the fully qualified class name of the service object.

For example, if the Servlet Whiteboard implementation is listening on port 80 on the machine www.acme.com and the Servlet object is registered with the pattern "/servlet", then the Servlet object's service method is called when the following URL is used from a web browser:

```
http://www.acme.com/servlet
```

The following table describes the properties that can be used by Servlets registered as Whiteboard services. Additionally, the common properties listed in Table 140.3 on page 792 are supported.

Table 140.4 Service properties for Servlet Whiteboard services.

Service Property	Type	Description
osgi.http.whiteboard.servlet.« asyncSupported HttpWhiteboardServletAsyncSupported	Boolean | String *optional*	Declares whether the servlet supports the asynchronous operation mode. Allowed values are true and false independent of case. Defaults to false. See HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED.
osgi.http.whiteboard.servlet.« errorPage HttpWhiteboardServletErrorPage	String+ *optional†*	Register the servlet as an error page for the error code and/or exception specified; the value may be a fully qualified exception type name or a three-digit HTTP status code in the range 400-599. Special values 4xx and 5xx can be used to match value ranges. Any value not being a three-digit number is assumed to be a fully qualified exception class name. See HTTP_WHITEBOARD_SERVLET_ERROR_PAGE.
osgi.http.whiteboard.servlet.« name HttpWhiteboardServletName	String *optional†*	The name of the servlet. This name is used as the value of the jakarta.servlet.ServletConfig.getServletName method and defaults to the fully qualified class name of the service object. See HTTP_WHITEBOARD_SERVLET_NAME.
osgi.http.whiteboard.servlet.« pattern HttpWhiteboardServletPattern	String+ *optional†*	Registration pattern(s) for the servlet. See HTTP_WHITEBOARD_SERVLET_PATTERN.
osgi.http.whiteboard.servlet.« multipart.enabled enabled	Boolean | String *optional*	Enables support for multipart configuration on the servlet. Allowed values are true and false independent of case. Defaults to false. See HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED.
osgi.http.whiteboard.servlet.« multipart.fileSizeThreshold fileSizeThreshold	Integer *optional*	The file size threshold after which the file is stored as a temporary file on disk while uploading. Defaults to *0*. Files will be stored in the directory as specified in ...location on the file system. See HTTP_WHITEBOARD_SERVLET_MULTIPART_FILESIZETHRESHOLD.

Service Property	Type	Description
osgi.http.whiteboard.servlet.« multipart.location location	String *optional*	The location where files are stored on disk. Defaults to the value of jakarta.servlet.context.tempdir servlet context attribute. If this attribute is not set, the value of the java.io.tmpdir system property will be used as default. If an absolute path is specified then this path is used as-is. If a relative path is specified, it will be used as relative to the default value. java.io.File.isAbsolute must be used to evaluate whether a path is absolute or relative. See HTTP_WHITEBOARD_SERVLET_MULTIPART_LOCATION.
osgi.http.whiteboard.servlet.« multipart.maxFileSize maxFileSize	Long *optional*	The maximum size for an uploaded file. Defaults to *unlimited*. Files larger than this size will cause a servlet exception. See HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXFILESIZE.
osgi.http.whiteboard.servlet.« multipart.maxRequestSize maxRequestSize	Long *optional*	The maximum size of a multipart/form-data request, in bytes. Defaults to *unlimited*. Requests larger than this value will cause a servlet exception. See HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXREQUESTSIZE.
servlet.init.*	String *optional*	Properties starting with this prefix are provided as init parameters to the jakarta.servlet.Servlet.init method. The servlet.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_SERVLET_INIT_PARAM_PREFIX.

† Note that at least one of the following properties *must* be specified on Servlet Whiteboard services:

```
osgi.http.whiteboard.servlet.pattern
osgi.http.whiteboard.servlet.name
osgi.http.whiteboard.servlet.errorPage
```

Servlet objects are initialized by a Servlet Whiteboard implementation before they start serving requests. The initialization is done by calling the Servlet object's Servlet.init(ServletConfig) method. The ServletConfig parameter provides access to the initialization parameters specified when the Servlet object was registered. Once the servlet is no longer used by the Servlet Whiteboard implementation the destroy method is called. Failure during Servlet.init will prevent the servlet from being used, which is reflected using a failure DTO. In such a case the system treats the servlet as unusable and attempts to find an alternative servlet matching the request.

If the service properties of the servlet Whiteboard service are modified, the destroy method is called. Subsequently the servlet is re-initialized. If a Prototype Service Factory is used for the servlet this re-initialization is done on a new service object.

When multiple Servlet Whiteboard implementations are present all of them can potentially process the Servlet. In such situations it can be useful to associate the servlet with a specific implementation by specifying the osgi.http.whiteboard.target property on the Servlet service to match its HttpServiceRuntime service.

If more than one Http Service Runtime matches the osgi.http.whiteboard.target property or the property is not set, the Servlet will be processed by all the matching implementations. A Servlet service that is processed by more than one Servlet Whiteboard implementation will have its init method called for each implementation that processes this Servlet. Similarly, the destroy method is called once when the Servlet is shut down once for each implementation that processed it. As multiple init and destroy calls on the same Servlet instance are generally not desirable, Servlet implementations should be registered as Prototype Service Factories as defined in the *OSGi Core Release 8*. This will ensure that each Servlet Whiteboard implementation processing the Servlet will use a separate instance, ensuring that only one init and destroy call is made per Servlet object. Servlets not registered as a Prototype Service Factory may received init and destroy calls multiple times on the same service object.

The following example code uses Declarative Services annotations to register a servlet whiteboard service.

```
@HttpWhiteboardServletPattern("/myservlet")
@Component(service = Servlet.class, scope = ServiceScope.PROTOTYPE,
    property = "servlet.init.myname=value")
public class MyServlet extends HttpServlet {
    private String name = "<not set>";

    public void init(ServletConfig config) {
        name = config.getInitParameter("myname");
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {
        resp.setContentType("text/plain");
        resp.getWriter().println("Servlet name: " + name);
    }
}
```

This example registers the servlet at: /myservlet. Requests for http://www.acme.com/myservlet map to the servlet, whose service method is called to process the request.

To associate the above example servlet with the example ServletContextHelper in *The Servlet Context* on page 786, add the following service property:

```
osgi.http.whiteboard.context.select=(osgi.http.whiteboard.context.name=my-context)
```

This will cause the servlet to move to http://www.acme.com/myapp/myservlet as configured by the custom Servlet Context Handler.

140.4.1 Multipart File Upload

Multipart file uploads are supported by specifying the osgi.http.whiteboard.servlet.multipart.* properties on the Servlet service registration. The following example illustrates this:

```
@Component(service = Servlet.class)
@HttpWhiteboardServletPattern("/image")
@HttpWhiteboardServletMultipart(enabled = true, maxFileSize = 200000)
public class ImageServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        Collection<Part> parts = request.getParts();

        for (Part part : parts) {
            System.out.printf("File %s, %s, %d%n", part.getName(),
                    part.getContentType(), part.getSize());

            try (InputStream is = part.getInputStream()) {
                // ...
            }
        }
    }
}
```

140.4.2 Error Pages

Servlets can be used to serve Error Pages. These are invoked when an exception is thrown during processing or if a servlet uses the sendError method with a status code between 400 and 599.

For a servlet service to handle error situations the service property osgi.http.whiteboard.servlet.errorPage must be set. This property can have multiple values, allowing a single servlet to handle a variety of error situations. Possible values are 3-digit HTTP error codes and fully qualified exception names.

Two special error code values are recognized. The special value 4xx means every error code in the 400-499 range. The special value 5xx means every error code in the 500-599 range. To override such wildcard error page for a specific error, register an error page with the specific error code and a higher service ranking. Error pages shadowed by other error pages are reported via the failure DTOs. A 4xx/5xx wildcard error page is only reported in the failure DTOs if it is shadowed by another wildcard page.

Matching exceptions follows the exception hierarchy. First the most specific exception class - the actual class of the exception - is looked up. If no matching error page for the most specific exception is found, the error page for the super class of the exception is looked up and so on. The process ends by looking up an error page for the java.lang.Throwable class.

While not being common practice, it is possible to combine the osgi.http.whiteboard.servlet.errorPage and osgi.http.whiteboard.servlet.pattern properties. If a single servlet registration has both these registration properties it is considered both an ordinary servlet as well as an error page.

If an error or exception occurs for which an error page servlet can be matched, it is invoked to render the error page. If the error page servlet causes an error or exception while handling the request, an implementation built-in error page is returned.

For example:

```
@Component(service = Servlet.class, scope = ServiceScope.PROTOTYPE)
@HttpWhiteboardServletErrorPage(errorPage = {"java.io.IOException", "500"})
public class MyErrorServlet extends HttpServlet {
   ...
}
```

The example servlet is invoked in case of a 500 error code, or if an IOException (or subclass) occurs. If there is more than one error page registered for the same exception or error code, the service that is first in ranking order, as specified in ServiceReference.compareTo, is selected as the handling servlet.

140.4.3 Asynchronous Request Handling

Servlets can use the asynchronous request handling feature, as defined by the servlet specification.

A servlet or servlet filter supporting the asynchronous mode must declare this with the appropriate service property osgi.http.whiteboard.servlet.asyncSupported or osgi.http.whiteboard.filter.asyncSupported.

An example simple asynchronous servlet that handles the servlet requests in a thread from a custom thread pool rather than in the thread provided by the servlet container:

```
@Component(service = Servlet.class, scope = ServiceScope.PROTOTYPE)
@HttpWhiteboardServletPattern("/as")
@HttpWhiteboardServletAsyncSupported
public class AsyncServlet extends HttpServlet {
    ExecutorService executor = Executors.newCachedThreadPool(
        r -> new Thread(r, "Pooled Thread"));

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {
        doGetAsync(req.startAsync());
    }

    private void doGetAsync(AsyncContext asyncContext) {
        executor.submit(() -> {
            try {
                PrintWriter writer = asyncContext.getResponse().getWriter();
                writer.print("Servlet executed async in: " +
                    Thread.currentThread()); // writes 'Pooled Thread'
            } finally {
                asyncContext.complete();
            }
            return null;
        });
    }
}
```

140.4.4 Annotations

Annotations defined in the Servlet API Specifications are ignored by an implementation of the Servlet Whiteboard Specification. The OSGi service model is used instead by this specification.

Implementations of this specification *may* support these annotations through a proprietary opt-in mechanism.

140.5 Registering Servlet Filters

Servlet filters provide a mechanism to intercept servlet invocations. They support modifying the ServletRequest and ServletResponse objects and are often used to augment web pages generated by servlets, for example with a common header or footer. Servlet filters can also be used to handle security, do logging or transform the content produced by a servlet to a certain format.

Similar to servlets, servlet filters are registered as Whiteboard services, by registering a jakarta.servlet.Filter instance in the Service Registry. The following table describes the supported service properties. In addition the common properties as described in Table 140.3 on page 792 are supported.

Table 140.5 Service properties for Filter Whiteboard services.

Service Property	Type	Description
osgi.http.whiteboard.filter.« asyncSupported HttpWhiteboardFilterAsyncSupported	String *optional*	Declares whether the servlet filter supports asynchronous operation mode. Allowed values are true and false independent of case. Defaults to false. See HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED.
osgi.http.whiteboard.filter.« dispatcher HttpWhiteboardFilterDispatcher	String+ *optional*	Select the dispatcher configuration when the servlet filter should be called. Allowed string values are REQUEST, ASYNC, ERROR, INCLUDE, and FORWARD. The default for a filter is REQUEST. See HTTP_WHITEBOARD_FILTER_DISPATCHER.

Service Property	Type	Description
osgi.http.whiteboard.filter.name HttpWhiteboardFilterName	String *optional*	The name of a servlet filter. This name is used as the value of the FilterConfig.getFilterName method and defaults to the fully qualified class name of the service object. See HTTP_WHITEBOARD_FILTER_NAME.
osgi.http.whiteboard.filter.pattern HttpWhiteboardFilterPattern	String+ *optional†*	Apply this servlet filter to the specified URL path patterns. The format of the patterns is specified in the servlet specification. See HTTP_WHITEBOARD_FILTER_PATTERN.
osgi.http.whiteboard.filter.regex HttpWhiteboardFilterRegex	String+ *optional†*	Apply this servlet filter to the specified URL paths. The paths are specified as regular expressions following the syntax defined in the java.util.regex.Pattern class. See HTTP_WHITEBOARD_FILTER_REGEX.
osgi.http.whiteboard.filter.servlet HttpWhiteboardFilterServlet	String+ *optional†*	Apply this servlet filter to the referenced servlet(s) by name. See HTTP_WHITEBOARD_FILTER_SERVLET.
filter.init.*	String+ *optional*	Properties starting with this prefix are passed as init parameters to the Filter.init method. The filter.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_FILTER_INIT_PARAM_PREFIX.

† Note that at least one of the following properties *must* be specified on Filter Whiteboard services:

```
osgi.http.whiteboard.filter.pattern
osgi.http.whiteboard.filter.regex
osgi.http.whiteboard.filter.servlet
```

Similar to servlets, Filter objects are initialized by a Servlet Whiteboard implementation before they start filtering requests. The initialization is done by calling the Filter.init(FilterConfig) method. The FilterConfig parameter provides access to filter.init.* properties on the servlet filter service registration. Once the Filter is no longer used by the Servlet Whiteboard implementation, the destroy method is called. When the service properties on the servlet filter are modified, the destroy method is called and the servlet filter is subsequently re-initialized, if it can still be associated with a Servlet Whiteboard implementation after the modification. By default, a servlet filter can be used with any Servlet Context Helper or Servlet Whiteboard implementation. To restrict a servlet filter to a single implementation or a specific Servlet Context Helper, the *Common Whiteboard Properties* on page 791 can be used.

To deal with the dynamicity of the Whiteboard service lifecycle, it is recommended to implement a servlet filter as Prototype Service Factory service. This will ensure that one single servlet filter instance only receives one init and one destroy call. Otherwise a single servlet filter instance can receive multiple such calls. This is similar to the behavior recommended for Servlet Whiteboard services.

Multiple servlet filters can process the same servlet request/response. If more than one Filter matches, they are processed in ranking order, as specified in ServiceReference.compareTo. The servlet filter with the highest ranking is processed first in the filter chain, while the servlet filter with the lowest ranking is processed last, before the Servlet.service method is called. After the servlet completes its service method the filter chain is unwound in reverse order.

Servlet filters are only applied to servlet requests if they are bound to the same Servlet Context Helper and the same Servlet Whiteboard implementation.

The example Filter below adds some text before and after the content generated by a servlet:

```
@Component(scope = ServiceScope.PROTOTYPE)
@HttpWhiteboardFilterPattern("/*")
public class MyFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {}
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        response.getWriter().write("before");
        chain.doFilter(request, response);
        response.getWriter().write("after");
    }

    public void destroy() {}
}
```

140.5.1 Servlet Pre-Processors

Servlet Filters are always run after handleSecurity is called. However in some cases it is necessary to process servlet requests before security is handled. For example if all requests must be logged, even ones that are rejected by security. In other scenarios, requests may need to be prepared for the handleSecurity call.

A whiteboard Preprocessor service can be registered to handle such cases. The Preprocessor service only supports the following service registration properties:

Table 140.6 Service properties for Preprocessor *Whiteboard services.*

Service Property	Type	Description
osgi.http.whiteboard.target HttpWhiteboardTarget	String *optional*	The value of this service property is an LDAP-style filter expression to select the Servlet Whiteboard implementation(s) to handle this Whiteboard service. The LDAP filter is used to match HttpServiceRuntime services. Each Servlet Whiteboard *implementation* exposes exactly one HttpServiceRuntime service. This property is used to associate the Whiteboard service with the Servlet Whiteboard implementation that registered the HttpServiceRuntime service. If this property is not specified, all Servlet Whiteboard implementations can handle the service. See HTTP_WHITEBOARD_TARGET.
preprocessor.init.*	String+ *optional*	Properties starting with this prefix are passed as init parameters to the Filter.init method. The preprocessor.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_PREPROCESSOR_INIT_PARAM_PREFIX.

A Preprocessor is invoked before request dispatching is performed. If multiple pre-processors are registered they are invoked in the order as described for servlet filters.

The Preprocessor has the same API as the servlet Filter and is handled in the same way, the init and destroy are called at the appropriate life-cycle events. However, as pre-processors are called before dispatching, the targeted servlet context is not yet know. Therefore the FilterConfig.getServletContext returns the servlet context of the backing implementation, the same context as returned by the request. As a pre-processor instance is not associated with a specific servlet context, it is safe to implement it as a singleton.

When called in the doFilter method, the pre-processor can use the FilterChain to invoke the next pre-processor, or if the end of the chain is reached, start processing the request. The pre-processor can also terminate the processing and generate a response directly. Before request processing returns to the pre-processors finishSecurity is called. If an exception is thrown during request processing, the exception is propagated through the pre-processors.

The example Preprocessor below logs a message before and after request processing:

```
@Component
public class MyPreprocessor implements Preprocessor {
    @Reference(service=LoggerFactory.class)
    private Logger logger;

    public void init(FilterConfig filterConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        logger.debug("Request processing starts");
        chain.doFilter(request, response);
        logger.debug("Request processing ends");
    }

    public void destroy() {}
}
```

140.6 Registering Resources

A resource is a file containing images, static HTML pages, JavaScript, CSS, sounds, movies, etc. Resources do not require any handling from the bundle. They are transferred directly from their source - usually the JAR file that contains the code for the bundle - to the requester.

Resources can be served by registering a service of any type with a service registration property that marks it as a resource service: osgi.http.whiteboard.resource.pattern. The actual service object registered is not used to serve resources, it is merely used to inform the Servlet Whiteboard implementation to serve resources from a certain source.

The following table describes the supported service properties. In addition the common properties as described in Table 140.3 on page 792 are supported.

Table 140.7 *Service properties for resource services.*

Service Property	Type	Description
osgi.http.whiteboard.resource.pattern pattern	String+ *required*	The pattern(s) to be used to serve resources. As defined by the [4] *Jakarta Servlet 5.0 Specification* in section 12.2, *Specification of Mappings*. This property marks the service as a resource service. See HTTP_WHITEBOARD_RESOURCE_PATTERN.
osgi.http.whiteboard.resource.prefix prefix	String *required*	The prefix used to map a requested resource to the bundle's entries. If the request's path info is not null, it is appended to this prefix. The resulting string is passed to the getResource(String) method of the associated Servlet Context Helper. See HTTP_WHITEBOARD_RESOURCE_PREFIX.

The examples below use Declarative Services annotations to register a resources service. Note that this service is purely used to convey information to the Servlet Whiteboard implementation and is never invoked.

```
@Component(service = MyResourceService.class)
@HttpWhiteboardResource(pattern = "/files/*", prefix = "/www")
public class MyResourceService {}
```

A Servlet Whiteboard implementation configured on port 80 will serve a request for http://localhost/files/cheese.html from the location /www/cheese.html.

The following example maps requests for /favicon.ico to serve the /logo.png resource. Note that the pattern is not appended to the prefix as the path info in this case is null.

```
@Component(service = SomeResourceService.class)
@HttpWhiteboardResource(pattern = "/favicon.ico", prefix = "/logo.png")
public class SomeResourceService {}
```

The above examples use the default ServletContextHelper implementation, which loads these resources from the bundle that registered the resource service. For more control around serving resources, a resources service can be associated to a custom ServletContextHelper. For example, a custom Servlet Context Helper can serve resources from locations other than the current bundle.

140.6.1 Overlapping Resource and Servlet Registrations

Resources and servlets registered with the same Servlet Context share a single URI namespace. This means that the value specified in osgi.http.whiteboard.resource.pattern competes with the osgi.http.whiteboard.servlet.pattern property specified on servlets. If these values overlap, the rules as outlined in *Registering Servlets* on page 792 are used to resolve conflicts, where resource services are treated just like servlets. Shadowed resource patterns are reported as FailedResourceDTO.

140.7 Registering Listeners

The servlet specification defines listener interfaces that can be implemented to receive a variety of servlet-related events. When using the Servlet Whiteboard implementation these listeners can be registered as Whiteboard services.

- ServletContextListener - Receive notifications when Servlet Contexts are initialized and destroyed.
- ServletContextAttributeListener - Receive notifications for Servlet Context attribute changes.
- ServletRequestListener - Receive notifications for servlet requests coming in and being destroyed.
- ServletRequestAttributeListener - Receive notifications when servlet Request attributes change.
- HttpSessionListener - Receive notifications when Http Sessions are created or destroyed.
- HttpSessionAttributeListener - Receive notifications when Http Session attributes change.
- HttpSessionIdListener - Receive notifications when Http Session ID changes.

Events are sent to listeners registered in the Service Registry with the osgi.http.whiteboard.listener service property set to true, independent of case. Listeners can be associated with a ServletContextHelper as described in *Common Whiteboard Properties* on page 791. Listeners not specifically associated with a Servlet Context Helper will receive events relating to the *default* Servlet Context Helper.

Multiple listeners of the same type registered with a given Servlet Context Helper are invoked in ranking order, as specified in ServiceReference.compareTo.

Table 140.8 Service properties for listener services.

Service Property	Type	Description
osgi.http.whiteboard.listener HttpWhiteboardListener	Boolean | String *required*	When set to true this listener service is handled by the Servlet Whiteboard implementation. When not set or set to false the service is ignored. Any other value is invalid and will be reflected in a FailedListenerDTO. Note the property value is case independent. See HTTP_WHITEBOARD_LISTENER.

An example listener that reports on client requests being initialized and destroyed is listed below:

```
@Component
@HttpWhiteboardListener
public class MyServletRequestListener implements ServletRequestListener {
    public void requestInitialized(ServletRequestEvent sre) {
        System.out.println("Request initialized for client: " +
            sre.getServletRequest().getRemoteAddr());
    }

    public void requestDestroyed(ServletRequestEvent sre) {
        System.out.println("Request destroyed for client: " +
            sre.getServletRequest().getRemoteAddr());
    }
}
```

For more details on the behavior of the listeners see the [4] *Jakarta Servlet 5.0 Specification.*

140.8 Life Cycle

If a Whiteboard service is used by a Servlet Whiteboard implementation, the following order of actions are performed:

1. The service is obtained from the service registry.
2. For servlets and servlet filters, init is called.

When the service is not used anymore, these actions are performed:

3. For servlets and servlet filters, destroy is called.
4. The service is released.

Note that some of the above actions may not be performed immediately, allowing an implementation to utilize lazy or asynchronous behavior.

As servlets and servlet filters services might come and go as well as ServletContextHelper services might come and go, use of the Whiteboard services can be very dynamic. Therefore servlet and servlet filter services might transition between bound to a Servlet Whiteboard implementation to being unbound and back to be bound. For example, when a matching Servlet Context Helper with the same name arrives with a higher ranking than the currently bound Servlet Context Helper, the servlet will be destroyed and re-initialized, bound to this better matching Servlet Context Helper. This is to ensure that timing issues cannot dictate the topology of the system.

As init and destroy are called each time the service life cycle changes, the recommended way to register services is to use the Prototype Service scope as defined in the *OSGi Core Release 8*. This ensures a new instance is created for each time such service is re-initialized. If the prototype scope is not used, the service should be prepared that after a call to destroy a new initialization through init might follow.

140.8.1 Whiteboard Service Dynamics and Active Requests

When the Servlet Whiteboard implementation receives a network request it establishes the processing pipeline based on the available Whiteboard services (servlets, servlet filters and resource services) and executes this pipeline. Between establishing the pipeline and finishing the processing, services used in this pipeline might become unregistered. It is up to the Servlet Whiteboard implementation whether it completes the active request or throws a Servlet Exception in this case.

140.9 The Http Service Runtime Service

The HttpServiceRuntime service represents the runtime state information of a Servlet Whiteboard implementation. This information is provided through Data Transfer Objects (DTOs). The architecture of OSGi DTOs is described in *OSGi Core Release 8*.

Each Servlet Whiteboard implementation registers exactly one HttpServiceRuntime service which can be used to target Whiteboard services defined in this specification to a specific Http Whiteboard implementation.

The HttpServiceRuntime provides service registration properties to declare its underlying Servlet Whiteboard implementation. These service properties can include implementation-specific key-value pairs. They also include the following:

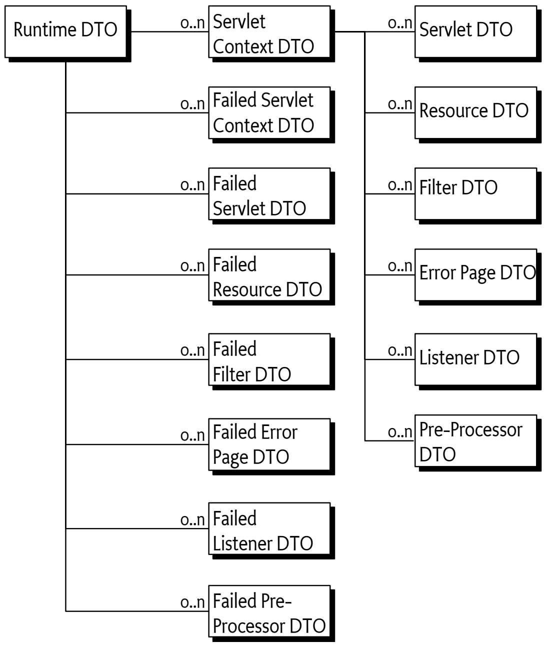
Table 140.9 Service properties for the HttpServiceRuntime service

Service Property	Type	Description
osgi.http.endpoint	String+	Endpoint(s) where this Servlet Whiteboard implementation is listening. Registered Whiteboard services are made available here. Values could be provided as URLs e.g. http://192.168.1.10:8080/ or relative paths, e.g. /myapp/. Relative paths may be used if the scheme and authority parts of the URLs are not known such as in a bridged Servlet Whiteboard implementation. If the Servlet Whiteboard implementation is serving the root context and scheme and authority are not known, the value of the property is /. Each entry must end with a slash. See HTTP_SERVICE_ENDPOINT.
service.changecount	Long	Whenever the DTOs available from the Http Service Runtime service change, the value of this property will increase by an amount of 1 or more. This allows interested parties to be notified of changes to the DTOs by observing Service Events of type MODIFIED for the HttpServiceRuntime service. See org.osgi.framework.Constants.SERVICE_CHANGECOUNT in *OSGi Core Release 8*.

The Http Service Runtime service provides information on registered Whiteboard services through the RuntimeDTO and RequestInfoDTO. The RuntimeDTO provides information on services that have been successfully registered as well as information about the Whiteboard services that were not successfully registered. Whiteboard services that have the required properties set but cannot be processed, are reflected in the failure DTOs. Whiteboard services of interfaces described in this specification that do not have the required properties set are ignored and not reflected in the failure DTOs.

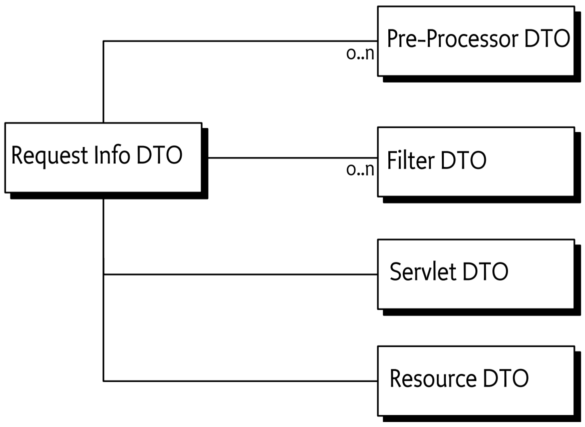
The Runtime DTO can be obtained using the getRuntimeDTO() method. The Runtime DTO provided has the following structure:

Figure 140.3 Runtime DTO Overview Diagram



Handlers for a given request path can be found with the calculateRequestInfoDTO(String) method. This method returns a RequestInfoDTO with the following structure:

Figure 140.4 Request Info DTO Overview Diagram



140.10 Configuration Properties

If the Servlet Whiteboard implementation does not have its port values configured through some other means, the implementation should use the following Framework properties to determine the port values to listen on.

- org.osgi.service.http.port - This property specifies the port used for servlets and resources accessible via HTTP. The default value for this property is 80.

- org.osgi.service.http.port.secure - This property specifies the port used for servlets and resources accessible via HTTPS. The default value for this property is 443.

140.11 Capabilities

140.11.1 osgi.implementation Capability

The Servlet Whiteboard implementation bundle must provide the osgi.implementation capability with name osgi.http. This capability can be used by provisioning tools and during resolution to ensure that a Servlet Whiteboard implementation is present to process the Whiteboard services defined in this specification. The capability must also declare a uses constraint for the Servlet and OSGi Servlet Whiteboard packages and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
       osgi.implementation="osgi.http";
       uses:="jakarta.servlet, jakarta.servlet.http,
              org.osgi.service.servlet.context,
              org.osgi.service.servlet.whiteboard";
       version:Version="2.0"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

Bundles registering services to be picked up by the Http Whiteboard implementation should require the osgi.implementation capability. For example:

```
Require-Capability: osgi.implementation;
       filter:="(&(osgi.implementation=osgi.http)
                 (version>=2.0)(!(version>=2.0)))"
```

To simplify the creation of this requirement, the RequireHttpWhiteboard annotation can be used.

140.11.2 osgi.contract Capability

The Servlet Whiteboard implementation must provide a capability in the osgi.contract namespace with name JakartaServlet if it exports the jakarta.servlet and jakarta.servlet.http packages. See [5] *Portable Java Contract Definitions*.

Providing the osgi.contract capability enables developer to build portable bundles for packages that are not versioned under OSGi Semantic Versioning rules. For more details see *osgi.contract Namespace* on page 709.

If the Servlet API is provided by another bundle, the Servlet Whiteboard implementation must be a consumer of the API and require the contract.

140.11.3 osgi.service Capability

The bundle providing the HttpServiceRuntime service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.servlet.runtime and org.osgi.service.servlet.runtime.dto packages:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.servlet.runtime.HttpServiceRuntime";
  uses:="org.osgi.service.servlet.runtime,org.osgi.service.servlet.runtime.dto"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

140.12 Security

This section only applies when executing in an OSGi environment which is enforcing Java permissions.

140.12.1 Service Permissions

Bundles that need to register Servlet Whiteboard services must be granted ServicePermission[interfaceName, REGISTER] where interface name is the Servlet Whiteboard service interface name.

The Servlet Whiteboard implementation must be granted ServicePermission[interfaceName, GET] to retrieve the Http Whiteboard services from the service registry.

140.12.2 Introspection

Bundles that need to introspect the state of the Http runtime will need ServicePermission[org.osgi.service.servlet.runtime.HttpServiceRuntime, GET] to obtain the HttpServiceRuntime service and access the DTO types.

140.12.3 Accessing Resources with the Default Servlet Context Helper Implementation

The Servlet Whiteboard implementation must be granted AdminPermission[*,RESOURCE] so that bundles may use the default ServletContextHelper implementation. This is necessary because the implementation of the default ServletContextHelper must call Bundle.getEntry to access the resources of a bundle and this method requires the caller to have AdminPermission[bundle,RESOURCE].

Any bundle may access resources in its own bundle by calling Class.getResource. This operation is privileged. The resulting URL object may then be passed to the Http Whiteboard implementation as the result of a ServletContextHelper.getResource call. No further permission checks are performed when accessing bundle entry or resource URL objects, so the Servlet Whiteboard implementation does not need to be granted any additional permissions.

140.12.4 Accessing Other Types of Resources

In order to access resources that were not returned from the default ServletContextHelper implementation, the Servlet Whiteboard implementation must be granted sufficient privileges to access these resources. For example, if the getResource method of a ServletContextHelper service returns a file URL, the Servlet Whiteboard implementation requires the corresponding FilePermission to read the file. Similarly, if the getResource method of a ServletContextHelper service returns an HTTP URL, the Servlet Whiteboard implementation requires the corresponding SocketPermission to connect to the resource.

Therefore, in most cases, the Servlet Whiteboard implementation should be a privileged service that is granted sufficient permission to serve any bundle's resources, no matter where these resources are located. Therefore, the Servlet Whiteboard implementation must capture the AccessControlContext object of the bundle registering a ServletContextHelper service, and then use the captured AccessControlContext object when accessing resources returned by the ServletContextHelper service. This situation prevents a bundle from supplying resources that it does not have permission to access.

Therefore, the Servlet Whiteboard implementation should follow a scheme like the following example. When using a ServletContextHelper service, it should capture the context.

```
ServiceReference<ServletContextHelper> servletContextHelperReference = ...
AccessControlContext acc = servletContextHelperReference.getBundle()
    .adapt(AccessControlContext.class);
```

When a URL returned by the getResource method of a ServletContextHelper service is used by the Servlet Whiteboard implementation, the implementation must use the URL in a doPrivileged construct using the AccessControlContext object of the registering bundle:

```
AccessController.doPrivileged(
    new PrivilegedExceptionAction() {
        public Object run() throws Exception {
        ...
        }
    }, acc);
```

This ensures the Servlet Whiteboard implementation can only use the URL if the bundle registering the ServletContextHelper service that returned the URL also has permission to use the URL. The use of a captured AccessControlContext only applies when accessing URL objects returned by the getResource method of the ServletContextHelper service.

140.12.5 Calling Servlet Whiteboard Services

This specification does not require that the Servlet Whiteboard implementation is granted All Permission or wraps calls to the Http Whiteboard services in a doPrivileged block. Therefore, it is the responsibility of the Servlet Whiteboard service implementations to use a doPrivileged block when performing privileged operations.

140.12.6 Multipart Upload

If multipart upload is enabled for a servlet, the uploaded data is usually temporarily written to a file. Therefore if security is enabled file permissions must be granted accordingly.

If a servlet is using the default path to store uploaded data, the Servlet Whiteboard implementation needs FilePermission[path, "read,write,delete"] for that path. As the servlet is reading the data, the bundle containing the servlet needs FilePermission[path, "read"] for that path.

If a servlet is providing the path to store uploaded data, the bundle containing the servlet needs FilePermission[path, "read,write,delete"] for that path. The Servlet Whiteboard implementation needs the same permissions for that path. Therefore, it is the responsibility of the Servlet Whiteboard service implementations to use a doPrivileged block when performing the write operation.

If security is enabled and any of the above required permissions is not granted, the multipart handling servlet is regarded invalid and will not be registered. This state is reflected in the error DTOs.

140.13 org.osgi.service.servlet.context

Http Context Package Version 2.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.servlet.context; version="[2.0,3.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.servlet.context; version="[2.0,2.1)"

140.13.1 Summary

- ServletContextHelper - Helper service for a servlet context used by a Servlet Whiteboard implementation to serve HTTP requests.

140.13.2 public abstract class ServletContextHelper

Helper service for a servlet context used by a Servlet Whiteboard implementation to serve HTTP requests.

This service defines methods that the Servlet Whiteboard implementation may call to get information for a request when dealing with whiteboard services.

Each ServletContextHelper is registered with a "osgi.http.whiteboard.context.name" service property containing a name to reference by servlets, servlet filters, resources, and listeners. If there is more than one ServletContextHelper registered with the same context name, the one that is first in the ranking order is active, the others are inactive.

A context is registered with the "osgi.http.whiteboard.context.path" service property to define a path under which all services registered with this context are reachable. If there is more than one ServletContextHelper registered with the same path, each duplicate context path is searched in ranking order until a matching servlet or resource is found.

Servlets, servlet filters, resources, and listeners services may be associated with a ServletContextHelper service with the "osgi.http.whiteboard.context.select" service property. If the referenced ServletContextHelper service does not exist or is currently not active, the whiteboard services for that ServletContextHelper are not active either.

If no ServletContextHelper service is associated, that is no "osgi.http.whiteboard.context.select" service property is configured for a whiteboard service, a default ServletContextHelper is used.

Those whiteboard services that are associated with the same ServletContextHelper object will share the same ServletContext object.

The behavior of the methods on the default ServletContextHelper is defined as follows:

- getMimeType - Always returns null.
- handleSecurity - Always returns true.
- getResource - Assumes the named resource is in the bundle of the whiteboard service, addressed from the root. This method calls the whiteboard service bundle's Bundle.getEntry method, and returns the appropriate URL to access the resource. On a Java runtime environment that supports permissions, the Servlet Whiteboard implementation needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].
- getResourcePaths - Assumes that the resources are in the bundle of the whiteboard service. This method calls Bundle.findEntries method, and returns the found entries. On a Java runtime environment that supports permissions, the Servlet Whiteboard implementation needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].
- getRealPath - Always returns null.

See Also HttpWhiteboardConstants.HTTP_WHITEBOARD_CONTEXT_NAME, HttpWhiteboardConstants.HTTP_WHITEBOARD_CONTEXT_PATH

Concurrency Thread-safe

140.13.2.1 public static final String AUTHENTICATION_TYPE = "org.osgi.service.http.authentication.type"

HttpServletRequest attribute specifying the scheme used in authentication. The value of the attribute can be retrieved by HttpServletRequest.getAuthType.

140.13.2.2 **public static final String AUTHORIZATION = "org.osgi.service.useradmin.authorization"**

HttpServletRequest attribute specifying the Authorization object obtained from the org.osgi.service.useradmin.UserAdmin service. The value of the attribute can be retrieved by HttpServletRequest.getAttribute(ServletContextHelper.AUTHORIZATION).

140.13.2.3 **public static final String REMOTE_USER = "org.osgi.service.http.authentication.remote.user"**

HttpServletRequest attribute specifying the name of the authenticated user. The value of the attribute can be retrieved by HttpServletRequest.getRemoteUser.

140.13.2.4 **public ServletContextHelper()**

□ Construct a new context helper.

If needed, the subclass will have to handle the association with a specific bundle.

140.13.2.5 **public ServletContextHelper(Bundle bundle)**

bundle The bundle to be associated with this context helper.

□ Construct a new context helper associated with the specified bundle.

140.13.2.6 **public void finishSecurity(HttpServletRequest request, HttpServletResponse response)**

request The HTTP request.

response The HTTP response.

□ Finishes the security context for the specified request.

Implementations of this service can implement this method to clean up resources which have been setup in handleSecurity(HttpServletRequest, HttpServletResponse).

This method is only called if handleSecurity(HttpServletRequest, HttpServletResponse) returned true for the specified request. This method is called once the pipeline finishes processing or if an exception is thrown from within the pipeline execution.

The default implementation of this method does nothing.

See Also handleSecurity(HttpServletRequest, HttpServletResponse)

140.13.2.7 **public String getMimeType(String name)**

name The name for which to determine the MIME type.

□ Maps a name to a MIME type.

Called by the Servlet Whiteboard implementation to determine the MIME type for the specified name. For whiteboard services, the Servlet Whiteboard implementation will call this method to support the ServletContext method getMimeType. For resource servlets, the Servlet Whiteboard implementation will call this method to determine the MIME type for the Content-Type header in the response.

Returns The MIME type (e.g. text/html) of the specified name or null to indicate that the Servlet Whiteboard implementation should determine the MIME type itself.

140.13.2.8 **public String getRealPath(String path)**

path The virtual path to be translated to a real path.

□ Gets the real path corresponding to the given virtual path.

Called by the Servlet Whiteboard implementation to support the ServletContext method getRealPath for whiteboard services.

Returns The real path, or null if the translation cannot be performed.

140.13.2.9 public URL getResource(String name)

name The name of the requested resource.

□ Maps a resource name to a URL.

Called by the Servlet Whiteboard implementation to map the specified resource name to a URL. For servlets, the Servlet Whiteboard implementation will call this method to support the `ServletContext` methods `getResource` and `getResourceAsStream`. For resources, the Servlet Whiteboard implementation will call this method to locate the named resource.

The context can control from where resources come. For example, the resource can be mapped to a file in the bundle's persistent storage area via `BundleContext.getDataFile(name).toURI().toURL()` or to a resource in the context's bundle via `getClass().getResource(name)`

Returns A URL that a Servlet Whiteboard implementation can use to read the resource or `null` if the resource does not exist.

140.13.2.10 public Set<String> getResourcePaths(String path)

path The partial path used to match the resources, which must start with a /.

□ Returns a directory-like listing of all the paths to resources within the web application whose longest sub-path matches the supplied path argument.

Called by the Servlet Whiteboard implementation to support the `ServletContext` method `getResourcePaths` for whiteboard services.

Returns A Set containing the directory listing, or `null` if there are no resources in the web application whose path begins with the supplied path.

140.13.2.11 public boolean handleSecurity(HttpServletRequest request, HttpServletResponse response) throws IOException

request The HTTP request.

response The HTTP response.

□ Handles security for the specified request.

The Servlet Whiteboard implementation calls this method prior to servicing the specified request. This method controls whether the request is processed in the normal manner or an error is returned.

If the request requires authentication and the `Authorization` header in the request is missing or not acceptable, then this method should set the `WWW-Authenticate` header in the response object, set the status in the response object to Unauthorized(401) and return `false`. See also RFC 2617: HTTP Authentication: Basic and Digest Access Authentication [http://www.ietf.org/rfc/rfc2617.txt].

If the request requires a secure connection and the `getScheme` method in the request does not return 'https' or some other acceptable secure protocol, then this method should set the status in the response object to Forbidden(403) and return `false`.

When this method returns `false`, the Servlet Whiteboard implementation will send the response back to the client, thereby completing the request. When this method returns `true`, the Servlet Whiteboard implementation will proceed with servicing the request.

If the specified request has been authenticated, this method must set the AUTHENTICATION_TYPE request attribute to the type of authentication used, and the REMOTE_USER request attribute to the remote user (request attributes are set using the `setAttribute` method on the request). If this method does not perform any authentication, it must not set these attributes.

If the authenticated user is also authorized to access certain resources, this method must set the AUTHORIZATION request attribute to the `Authorization` object obtained from the `org.osgi.service.useradmin.UserAdmin` service.

The servlet responsible for servicing the specified request determines the authentication type and remote user by calling the `getAuthType` and `getRemoteUser` methods, respectively, on the request.

If there is the need to clean up resources at the end of the request, the method finishSecurity(HttpServletRequest, HttpServletResponse) can be implemented. That method is only called if this method returns `true`.

Returns `true` if the request should be serviced, `false` if the request should not be serviced and Servlet Whiteboard implementation will send the response back to the client.

Throws `IOException`– May be thrown by this method. If this occurs, the Servlet Whiteboard implementation will terminate the request and close the socket.

See Also finishSecurity(HttpServletRequest, HttpServletResponse)

140.14 org.osgi.service.servlet.runtime

Http Runtime Package Version 2.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.servlet.runtime; version="[2.0,3.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.servlet.runtime; version="[2.0,2.1)"`

140.14.1 Summary

- `HttpServiceRuntime` - The HttpServiceRuntime service represents the runtime information of a Servlet Whiteboard implementation.
- `HttpServiceRuntimeConstants` - Defines standard names for Http Runtime Service constants.

140.14.2 public interface HttpServiceRuntime

The HttpServiceRuntime service represents the runtime information of a Servlet Whiteboard implementation.

It provides access to DTOs representing the current state of the service.

The HttpServiceRuntime service must be registered with the HttpServiceRuntimeConstants.HTTP_SERVICE_ENDPOINT service property.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

140.14.2.1 public RequestInfoDTO calculateRequestInfoDTO(String path)

path The request path, relative to the root of the Servlet Whiteboard implementation.

□ Return a request info DTO containing the services involved with processing a request for the specified path.

Returns The request info DTO for the specified path.

140.14.2.2 public RuntimeDTO getRuntimeDTO()

□ Return the runtime DTO representing the current state.

Returns The runtime DTO.

140.14.3 public final class HttpServiceRuntimeConstants

Defines standard names for Http Runtime Service constants.

140.14.3.1 public static final String HTTP_SERVICE_ENDPOINT = "osgi.http.endpoint"

Http Runtime Service service property specifying the endpoints upon which the Servlet Whiteboard implementation is listening.

An endpoint value is a URL or a relative path, to which the Servlet Whiteboard implementation is listening. For example, `http://192.168.1.10:8080/` or `/myapp/`. A relative path may be used if the scheme and authority parts of the URL are not known, e.g. in a bridged Servlet Whiteboard implementation. If the Servlet Whiteboard implementation is serving the root context and neither scheme nor authority is known, the value of the property is "/". Both, a URL and a relative path, must end with a slash.

An Servlet Whiteboard implementation can be listening on multiple endpoints.

The value of this service property must be of type `String`, `String[]`, or `Collection<String>`.

140.15 org.osgi.service.servlet.runtime.dto

Http Runtime DTO Package Version 2.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.servlet.runtime.dto; version="[2.0,3.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.servlet.runtime.dto; version="[2.0,3.1)"`

140.15.1 Summary

- `BaseServletDTO` - Represents common information about a `jakarta.servlet.Servlet` service.
- `DTOConstants` - Defines standard constants for the DTOs.
- `ErrorPageDTO` - Represents a `jakarta.servlet.Servlet` for handling errors and currently being used by a servlet context.
- `FailedErrorPageDTO` - Represents a `jakarta.servlet.Servlet` service registered as an error page but currently not being used by a servlet context due to a problem.
- `FailedFilterDTO` - Represents a servlet `Filter` service which is currently not being used by a servlet context due to a problem.
- `FailedListenerDTO` - Represents a listener service which is currently not being used by a servlet context due to a problem.
- `FailedPreprocessorDTO` - Represents a preprocessor service which is currently not being used due to a problem.
- `FailedResourceDTO` - Represents a resource definition which is currently not being used by a servlet context due to a problem.
- `FailedServletContextDTO` - Represents a servlet context that is currently not used due to some problem.

- FailedServletDTO - Represents a jakarta.servlet.Servlet service which is currently not being used by a servlet context due to a problem.
- FilterDTO - Represents a servlet jakarta.servlet.Filter service currently being used for by a servlet context.
- ListenerDTO - Represents a listener currently being used by a servlet context.
- PreprocessorDTO - Represents a preprocessor org.osgi.service.servlet.whiteboard.Preprocessor service currently being used during request processing.
- RequestInfoDTO - Represents the services used to process a specific request.
- ResourceDTO - Represents a resource definition currently being used by a servlet context.
- RuntimeDTO - Represents the state of a Http Service Runtime.
- ServletContextDTO - Represents a jakarta.servlet.ServletContext created for servlets, resources, servlet Filters, and listeners associated with that servlet context.
- ServletDTO - Represents a jakarta.servlet.Servlet currently being used by a servlet context.

140.15.2 public abstract class BaseServletDTO extends DTO

Represents common information about a jakarta.servlet.Servlet service.

Concurrency Not Thread-safe

140.15.2.1 public boolean asyncSupported

Specifies whether the servlet supports asynchronous processing.

140.15.2.2 public Map<String, String> initParams

The servlet initialization parameters as provided during registration of the servlet. Additional parameters like the Http Service Runtime attributes are not included. If the service has no initialization parameters, the map is empty.

140.15.2.3 public String name

The name of the servlet. This value is never null, unless this object represents a FailedServletDTO or a FailedErrorPageDTO where the value might be null.

140.15.2.4 public long serviceId

Service property identifying the servlet. In the case of a servlet registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.2.5 public long servletContextId

The service id of the servlet context for the servlet represented by this DTO.

140.15.2.6 public String servletInfo

The information string from the servlet.

This is the value returned by the Servlet.getServletInfo() method. For a FailedServletDTO or a FailedErrorPageDTO this is always null.

140.15.2.7 public BaseServletDTO()

140.15.3 public final class DTOConstants

Defines standard constants for the DTOs.

140.15.3.1 public static final int FAILURE_REASON_EXCEPTION_ON_INIT = 4

An exception occurred during initializing of the service.

This reason can only happen for servlets and servlet filters.

140.15.3.2 public static final int FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING = 1

No matching `org.osgi.service.servlet.context.ServletContextHelper`.

140.15.3.3 public static final int FAILURE_REASON_SERVICE_IN_USE = 7

The service is not registered as a prototype scoped service and is already in use with a servlet context and therefore can't be used with another servlet context.

140.15.3.4 public static final int FAILURE_REASON_SERVICE_NOT_GETTABLE = 5

The service is registered in the service registry but getting the service fails as it returns `null`.

140.15.3.5 public static final int FAILURE_REASON_SERVLET_CONTEXT_FAILURE = 2

Matching `org.osgi.service.servlet.context.ServletContextHelper`, but the context is not used due to a problem with the context.

140.15.3.6 public static final int FAILURE_REASON_SERVLET_READ_FROM_DEFAULT_DENIED = 10

The servlet is not registered as it is configured to have multipart enabled, but the bundle containing the servlet has no read permission to the default location for the uploaded files.

140.15.3.7 public static final int FAILURE_REASON_SERVLET_WRITE_TO_LOCATION_DENIED = 8

The servlet is not registered as it is configured to have multipart enabled, but the bundle containing the servlet has no write permission to the provided location for the uploaded files.

140.15.3.8 public static final int FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE = 3

Service is shadowed by another service.

For example, another service with the same service properties but having a higher service ranking. See org.osgi.framework.ServiceReference.compareTo(Object).

140.15.3.9 public static final int FAILURE_REASON_UNKNOWN = 0

Failure reason is unknown.

140.15.3.10 public static final int FAILURE_REASON_VALIDATION_FAILED = 6

The service is registered in the service registry but the service properties are invalid.

140.15.3.11 public static final int FAILURE_REASON_WHITEBOARD_WRITE_TO_DEFAULT_DENIED = 9

The servlet is not registered as it is configured to have multipart enabled, but the whiteboard implementation has no write permission to the default location for the uploaded files.

140.15.3.12 public static final int FAILURE_REASON_WHITEBOARD_WRITE_TO_LOCATION_DENIED = 11

The servlet is not registered as it is configured to have multipart enabled, but the whiteboard implementation has no write permission to the provided location for the uploaded files.

140.15.4 public class ErrorPageDTO extends BaseServletDTO

Represents a `jakarta.servlet.Servlet` for handling errors and currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.4.1 public long[] errorCodes

The error codes the error page is used for. This array might be empty.

140.15.4.2 public String[] exceptions

The exceptions the error page is used for. This array might be empty.

140.15.4.3 public ErrorPageDTO()

140.15.5 public class FailedErrorPageDTO extends ErrorPageDTO

Represents a jakarta.servlet.Servlet service registered as an error page but currently not being used by a servlet context due to a problem.

As the servlet represented by this DTO is not used due to a failure, the field FailedErrorPageDTO.servletContextId always returns 0 and does not point to an existing org.osgi.service.servlet.context.ServletContextHelper.

Concurrency Not Thread-safe

140.15.5.1 public int failureReason

The reason why the servlet represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN, DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT, DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING, DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE, DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE, DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.5.2 public FailedErrorPageDTO()

140.15.6 public class FailedFilterDTO extends FilterDTO

Represents a servlet Filter service which is currently not being used by a servlet context due to a problem.

As the service represented by this DTO is not used due to a failure, the field FailedFilterDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.6.1 public int failureReason

The reason why the servlet filter represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN, DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT, DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING, DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE, DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE, DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.6.2 public FailedFilterDTO()

140.15.7 public class FailedListenerDTO extends ListenerDTO

Represents a listener service which is currently not being used by a servlet context due to a problem.

As the listener represented by this DTO is not used due to a failure, the field FailedErrorPageDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.7.1 public int failureReason

The reason why the listener represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN, DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT, DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING, DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE, DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE, DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.7.2 public FailedListenerDTO()

140.15.8 public class FailedPreprocessorDTO extends PreprocessorDTO

Represents a preprocessor service which is currently not being used due to a problem.

Concurrency Not Thread-safe

140.15.8.1 public int failureReason

The reason why the preprocessor represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN, DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT, DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE

140.15.8.2 public FailedPreprocessorDTO()

140.15.9 public class FailedResourceDTO extends ResourceDTO

Represents a resource definition which is currently not being used by a servlet context due to a problem.

As the resource represented by this DTO is not used due to a failure, the field FailedResourceDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.9.1 public int failureReason

The reason why the resource represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN, DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT, DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING, DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,

DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.9.2 public FailedResourceDTO()

140.15.10 public class FailedServletContextDTO extends ServletContextDTO

Represents a servlet context that is currently not used due to some problem. The following fields return an empty array for a `FailedServletContextDTO`:

- ServletContextDTO.servletDTOs
- ServletContextDTO.resourceDTOs
- ServletContextDTO.filterDTOs
- ServletContextDTO.errorPageDTOs
- ServletContextDTO.listenerDTOs

The method ServletContextDTO.attributes returns an empty map for a `FailedServletContextDTO`.

Concurrency Not Thread-safe

140.15.10.1 public int failureReason

The reason why the servlet context represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.10.2 public FailedServletContextDTO()

140.15.11 public class FailedServletDTO extends ServletDTO

Represents a `jakarta.servlet.Servlet` service which is currently not being used by a servlet context due to a problem.

As the servlet represented by this DTO is not used due to a failure, the field FailedServletDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.11.1 public int failureReason

The reason why the servlet represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE,
DTOConstants.FAILURE_REASON_SERVLET_WRITE_TO_LOCATION_DENIED,
DTOConstants.FAILURE_REASON_WHITEBOARD_WRITE_TO_DEFAULT_DENIED,
DTOConstants.FAILURE_REASON_SERVLET_READ_FROM_DEFAULT_DENIED

140.15.11.2 public FailedServletDTO()

140.15.12 public class FilterDTO extends DTO

Represents a servlet `jakarta.servlet.Filter` service currently being used for by a servlet context.

Concurrency Not Thread-safe

140.15.12.1 public boolean asyncSupported

Specifies whether the servlet filter supports asynchronous processing.

140.15.12.2 public String[] dispatcher

The dispatcher associations for the servlet filter.

The specified names are used to determine in what occasions the servlet filter is called. This array is never `null`.

140.15.12.3 public Map<String, String> initParams

The servlet filter initialization parameters as provided during registration of the servlet filter. Additional parameters like the Http Service Runtime attributes are not included. If the servlet filter has not initialization parameters, this map is empty.

140.15.12.4 public String name

The name of the servlet filter. This field is never `null`.

140.15.12.5 public String[] patterns

The request mappings for the servlet filter.

The specified patterns are used to determine whether a request is mapped to the servlet filter. This array might be empty.

140.15.12.6 public String[] regexs

The request mappings for the servlet filter.

The specified regular expressions are used to determine whether a request is mapped to the servlet filter. This array might be empty.

140.15.12.7 public long serviceId

Service property identifying the servlet filter. In the case of a servlet filter registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet filter has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.12.8 public long servletContextId

The service id of the servlet context for the servlet filter represented by this DTO.

140.15.12.9 public String[] servletNames

The servlet names for the servlet filter.

The specified names are used to determine the servlets whose requests are mapped to the servlet filter. This array might be empty.

140.15.12.10 public FilterDTO()

140.15.13 public class ListenerDTO extends DTO

Represents a listener currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.13.1 public long serviceId

Service property identifying the listener. In the case of a Listener registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the listener has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.13.2 public long servletContextId

The service id of the servlet context for the listener represented by this DTO.

140.15.13.3 public String[] types

The fully qualified type names the listener. This array is never empty.

140.15.13.4 public ListenerDTO()

140.15.14 public class PreprocessorDTO extends DTO

Represents a preprocessor `org.osgi.service.servlet.whiteboard.Preprocessor` service currently being used during request processing.

Concurrency Not Thread-safe

140.15.14.1 public Map<String, String> initParams

The preprocessor initialization parameters as provided during registration of the preprocessor. Additional parameters like the Http Service Runtime attributes are not included. If the preprocessor has not initialization parameters, this map is empty.

140.15.14.2 public long serviceId

Service property identifying the preprocessor. In the case of a preprocessor registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the preprocessor has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.14.3 public PreprocessorDTO()

140.15.15 public class RequestInfoDTO extends DTO

Represents the services used to process a specific request.

Concurrency Not Thread-safe

140.15.15.1 public FilterDTO[] filterDTOs

The servlet filters processing this request. If no servlet filters are called for processing this request, an empty array is returned.

140.15.15.2 public String path

The path of the request relative to the root.

140.15.15.3 public ResourceDTO resourceDTO

The resource processing this request. If the request is processed by a resource, this field points to the DTO of the resource. If the request is processed by another type of component like a servlet, this field is null.

140.15.15.4 public long servletContextId

The service id of the servlet context processing the request represented by this DTO.

140.15.15.5 public ServletDTO servletDTO

The servlet processing this request. If the request is processed by a servlet, this field points to the DTO of the servlet. If the request is processed by another type of component like a resource, this field is null.

140.15.15.6 public RequestInfoDTO()

140.15.16 public class ResourceDTO extends DTO

Represents a resource definition currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.16.1 public String[] patterns

The request mappings for the resource.

The specified patterns are used to determine whether a request is mapped to the resource. This value is never null.

140.15.16.2 public String prefix

The prefix of the resource.

140.15.16.3 public long serviceId

Service property identifying the resource. In the case of a resource registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the resource has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.16.4 public long servletContextId

The service id of the servlet context for the resource represented by this DTO.

140.15.16.5 public ResourceDTO()

140.15.17 public class RuntimeDTO extends DTO

Represents the state of a Http Service Runtime.

Concurrency Not Thread-safe

140.15.17.1 public FailedErrorPageDTO[] failedErrorPageDTOs

Returns the representations of the error page jakarta.servlet.Servlet services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.2 public FailedFilterDTO[] failedFilterDTOs

Returns the representations of the jakarta.servlet.Filter services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.3 public FailedListenerDTO[] failedListenerDTOs

Returns the representations of the listeners associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.4 public FailedPreprocessorDTO[] failedPreprocessorDTOs

Returns the representations of the servlet org.osgi.service.servlet.whiteboard.Preprocessor services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.5 public FailedResourceDTO[] failedResourceDTOs

Returns the representations of the resources associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.6 public FailedServletContextDTO[] failedServletContextDTOs

Returns the representations of the jakarta.servlet.ServletContext objects currently not used by the Http service runtime due to some problem. The returned array may be empty.

140.15.17.7 public FailedServletDTO[] failedServletDTOs

Returns the representations of the jakarta.servlet.Servlet services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.17.8 public PreprocessorDTO[] preprocessorDTOs

Returns the representations of the org.osgi.service.servlet.whiteboard.Preprocessor objects used by the Http Service Runtime. The returned array may be empty if the Http Service Runtime is currently not using any org.osgi.service.servlet.whiteboard.Preprocessor objects.

140.15.17.9 public ServiceReferenceDTO serviceDTO

The DTO for the corresponding org.osgi.service.servlet.runtime.HttpServiceRuntime. This value is never null.

140.15.17.10 public ServletContextDTO[] servletContextDTOs

Returns the representations of the jakarta.servlet.ServletContext objects used by the Http Service Runtime. The returned array may be empty if the Http Service Runtime is currently not using any jakarta.servlet.ServletContext objects.

140.15.17.11 public RuntimeDTO()

140.15.18 public class ServletContextDTO extends DTO

Represents a jakarta.servlet.ServletContext created for servlets, resources, servlet Filters, and listeners associated with that servlet context. The Servlet Context is usually backed by a ServletContextHelper service.

Concurrency Not Thread-safe

140.15.18.1 public Map<String, Object> attributes

The servlet context attributes.

The value type must be a numerical type, Boolean, String, DTO or an array of any of the former. Therefore this method will only return the attributes of the servlet context conforming to this constraint. Other attributes are omitted. If there are no attributes conforming to the constraint, an empty map is returned.

140.15.18.2 public String contextPath

The servlet context path. This is the value returned by the ServletContext.getContextPath() method.

140.15.18.3 public ErrorPageDTO[] errorPageDTOs

Returns the representations of the error page Servlet services associated with this context. The representations of the error page Servlet services associated with this context. The returned array may be empty if this context is currently not associated with any error pages.

140.15.18.4 public FilterDTO[] filterDTOs

Returns the representations of the servlet Filter services associated with this context. The representations of the servlet Filter services associated with this context. The returned array may be empty if this context is currently not associated with any servlet Filter services.

140.15.18.5 public Map<String, String> initParams

The servlet context initialization parameters. This is the set of parameters provided when registering this context. Additional parameters like the Http Service Runtime attributes are not included. If the context has no initialization parameters, this map is empty.

140.15.18.6 public ListenerDTO[] listenerDTOs

Returns the representations of the listener services associated with this context. The representations of the listener services associated with this context. The returned array may be empty if this context is currently not associated with any listener services.

140.15.18.7 public String name

The name of the servlet context. The name of the corresponding ServletContextHelper.

This is the value returned by the ServletContext.getServletContextName() method.

140.15.18.8 public ResourceDTO[] resourceDTOs

Returns the representations of the resource services associated with this context. The representations of the resource services associated with this context. The returned array may be empty if this context is currently not associated with any resource services.

140.15.18.9 public long serviceId

Service property identifying the servlet context. In the case of a servlet context backed by a org.osgi.service.servlet.context.ServletContextHelper registered in the service registry and picked up by a Servlet Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet context is not backed by a service registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.18.10 public ServletDTO[] servletDTOs

Returns the representations of the Servlet services associated with this context. The representations of the Servlet services associated with this context. The returned array may be empty if this context is currently not associated with any Servlet services.

140.15.18.11 public ServletContextDTO()

140.15.19 public class ServletDTO extends BaseServletDTO

Represents a `jakarta.servlet.Servlet` currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.19.1 public boolean multipartEnabled

Specifies whether multipart support is enabled.

140.15.19.2 public int multipartFileSizeThreshold

Specifies the size threshold after which the file will be written to disk. If multipart is not enabled for this servlet, 0 is returned.

See Also multipartEnabled

140.15.19.3 public String multipartLocation

Specifies the location where the files can be stored on disk. If multipart is not enabled for this servlet, `null` is returned.

See Also multipartEnabled

140.15.19.4 public long multipartMaxFileSize

Specifies the maximum size of a file being uploaded. If multipart is not enabled for this servlet, 0 is returned.

See Also multipartEnabled

140.15.19.5 public long multipartMaxRequestSize

Specifies the maximum request size. If multipart is not enabled for this servlet, 0 is returned.

See Also multipartEnabled

140.15.19.6 public String[] patterns

The request mappings for the servlet.

The specified patterns are used to determine whether a request is mapped to the servlet. This array is never `null`. It might be empty for named servlets.

140.15.19.7 public ServletDTO()

140.16 org.osgi.service.servlet.whiteboard

Servlet Whiteboard Package Version 2.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.servlet.whiteboard; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.servlet.whiteboard; version="[2.0,2.1)"
```

140.16.1 Summary

- `HttpWhiteboardConstants` - Defines standard constants for the Servlet Whiteboard services.
- `Preprocessor` - Services registered as a `Preprocessor` using a whiteboard pattern are executed for every request before the dispatching is performed.

140.16.2 public final class HttpWhiteboardConstants

Defines standard constants for the Servlet Whiteboard services.

140.16.2.1 public static final String DISPATCHER_ASYNC = "ASYNC"

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied in the asynchronous context.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.2 public static final String DISPATCHER_ERROR = "ERROR"

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied when an error page is called.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.3 public static final String DISPATCHER_FORWARD = "FORWARD"

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to forward calls to the dispatcher.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.4 public static final String DISPATCHER_INCLUDE = "INCLUDE"

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to include calls to the dispatcher.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.5 public static final String DISPATCHER_REQUEST = "REQUEST"

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to client requests.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.6 public static final String HTTP_WHITEBOARD_CONTEXT_INIT_PARAM_PREFIX = "context.init."

Service property prefix referencing a ServletContextHelper service.

For ServletContextHelper services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated servlet context. The prefix is removed from the service property name to build the initialization parameter name.

For ServletContextHelper services, the value of each initialization parameter service property must be of type `String`.

140.16.2.7 public static final String HTTP_WHITEBOARD_CONTEXT_NAME = "osgi.http.whiteboard.context.name"

Service property specifying the name of an ServletContextHelper service.

For ServletContextHelper services, this service property must be specified. Context services without this service property are ignored.

Servlet, listener, servlet filter, and resource services might refer to a specific ServletContextHelper service referencing the name with the HTTP_WHITEBOARD_CONTEXT_SELECT property.

For ServletContextHelper services, the value of this service property must be of type String. The value must follow the "symbolic-name" specification from Section 1.3.2 of the OSGi Core Specification.

See Also HTTP_WHITEBOARD_CONTEXT_PATH, HTTP_WHITEBOARD_CONTEXT_SELECT, HTTP_WHITEBOARD_DEFAULT_CONTEXT_NAME

140.16.2.8 public static final String HTTP_WHITEBOARD_CONTEXT_PATH = "osgi.http.whiteboard.context.path"

Service property specifying the path of an ServletContextHelper service.

For ServletContextHelper services this service property is required. Context services without this service property are ignored.

This property defines a context path under which all whiteboard services associated with this context are registered. Having different contexts with different paths allows to separate the URL space.

For ServletContextHelper services, the value of this service property must be of type String. The value is either a slash for the root or it must start with a slash but not end with a slash. Valid characters are defined in rfc3986#section-3.3. Contexts with an invalid path are ignored.

See Also HTTP_WHITEBOARD_CONTEXT_NAME, HTTP_WHITEBOARD_CONTEXT_SELECT

140.16.2.9 public static final String HTTP_WHITEBOARD_CONTEXT_SELECT = "osgi.http.whiteboard.context.select"

Service property referencing a ServletContextHelper service.

For servlet, listener, servlet filter, or resource services, this service property refers to the associated ServletContextHelper service. The value of this property is a filter expression which is matched against the service registration properties of the ServletContextHelper service. If this service property is not specified, the default context is used. If there is no context service matching, the servlet, listener, servlet filter, or resource service is ignored.

For example, if a whiteboard service wants to select a servlet context helper with the name "Admin" the expression would be "(osgi.http.whiteboard.context.name=Admin)". Selecting all contexts could be done with "(osgi.http.whiteboard.context.name=*)".

For servlet, listener, servlet filter, or resource services, the value of this service property must be of type String.

See Also HTTP_WHITEBOARD_CONTEXT_NAME, HTTP_WHITEBOARD_CONTEXT_PATH

140.16.2.10 public static final String HTTP_WHITEBOARD_DEFAULT_CONTEXT_NAME = "default"

The name of the default ServletContextHelper. If a service is registered with this property, it is overriding the default context with a custom provided context.

See Also HTTP_WHITEBOARD_CONTEXT_NAME

140.16.2.11 public static final String HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED = "osgi.http.whiteboard.filter.asyncSupported"

Service property specifying whether a servlet Filter service supports asynchronous processing.

By default servlet filters services do not support asynchronous processing.

The value of this service property must be of type Boolean.

See Also Jakarta Servlet Specification Version 5.0, Asynchronous Processing [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#asynchronous-processing]

140.16.2.12 public static final String HTTP_WHITEBOARD_FILTER_DISPATCHER = "osgi.http.whiteboard.filter.dispatcher"

Service property specifying the dispatcher handling of a servlet Filter.

By default servlet filter services are associated with client requests only (see value DISPATCHER_REQUEST).

The value of this service property must be of type String, String[], or Collection<String>. Allowed values are DISPATCHER_ASYNC, DISPATCHER_ERROR, DISPATCHER_FORWARD, DISPATCHER_INCLUDE, DISPATCHER_REQUEST.

See Also Jakarta Servlet Specification Version 5.0, Filters and the RequestDispatcher [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#filters-and-the-requestdispatcher]

140.16.2.13 public static final String HTTP_WHITEBOARD_FILTER_INIT_PARAM_PREFIX = "filter.init."

Service property prefix referencing a Filter service.

For Filter services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated filter config. The prefix is removed from the service property name to build the initialization parameter name.

For Filter services, the value of each initialization parameter service property must be of type String.

140.16.2.14 public static final String HTTP_WHITEBOARD_FILTER_NAME = "osgi.http.whiteboard.filter.name"

Service property specifying the servlet filter name of a Filter service.

This name is used as the value for the FilterConfig.getFilterName() method. If this service property is not specified, the fully qualified name of the service object's class is used as the servlet filter name.

Servlet filter names should be unique among all servlet filter services associated with a single ServletContextHelper.

The value of this service property must be of type String.

140.16.2.15 public static final String HTTP_WHITEBOARD_FILTER_PATTERN = "osgi.http.whiteboard.filter.pattern"

Service property specifying the request mappings for a Filter service.

The specified patterns are used to determine whether a request should be mapped to the servlet filter. Filter services without this service property or the HTTP_WHITEBOARD_FILTER_SERVLET or the HTTP_WHITEBOARD_FILTER_REGEX service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also Jakarta Servlet Specification Version 5.0, Specification of Mappings [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#specification-of-mappings]

140.16.2.16 public static final String HTTP_WHITEBOARD_FILTER_REGEX = "osgi.http.whiteboard.filter.regex"

Service property specifying the request mappings for a servlet Filter service.

The specified regular expressions are used to determine whether a request should be mapped to the servlet filter. The regular expressions must follow the syntax defined in java.util.regex.Pattern. Servlet filter services without this service property or the HTTP_WHITEBOARD_FILTER_SERVLET or the HTTP_WHITEBOARD_FILTER_PATTERN service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also java.util.regex.Pattern

140.16.2.17 public static final String HTTP_WHITEBOARD_FILTER_SERVLET = "osgi.http.whiteboard.filter.servlet"

Service property specifying the servlet names for a servlet Filter service.

The specified names are used to determine the servlets whose requests should be mapped to the servlet filter. Servlet filter services without this service property or the

HTTP_WHITEBOARD_FILTER_PATTERN or the HTTP_WHITEBOARD_FILTER_REGEX service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

140.16.2.18 public static final String HTTP_WHITEBOARD_IMPLEMENTATION = "osgi.http"

The name of the implementation capability for the Servlet Whiteboard specification

140.16.2.19 public static final String HTTP_WHITEBOARD_LISTENER = "osgi.http.whiteboard.listener"

Service property to mark a Listener service as a Whiteboard service. Listener services with this property set to the string value "true" will be treated as Whiteboard services opting in to being handled by the Servlet Whiteboard implementation. If the value "false" is specified, the service is opting out and this case is treated exactly the same as if this property is missing. If an invalid value is specified this is treated as a failure.

The value of this service property must be of type String. Valid values are "true" and "false" ignoring case.

140.16.2.20 public static final String HTTP_WHITEBOARD_PREPROCESSOR_INIT_PARAM_PREFIX = "preprocessor.init."

Service property prefix referencing a Preprocessor service.

For Preprocessor services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated filter configuration. The prefix is removed from the service property name to build the initialization parameter name.

For Preprocessor services, the value of each initialization parameter service property must be of type String.

140.16.2.21 public static final String HTTP_WHITEBOARD_RESOURCE_PATTERN = "osgi.http.whiteboard.resource.pattern"

Service property specifying the request mappings for resources.

The specified patterns are used to determine whether a request should be mapped to resources. Resource services without this service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also Jakarta Servlet Specification Version 5.0, Specification of Mappings [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#specification-of-mappings], HTTP_WHITEBOARD_RESOURCE_PREFIX

140.16.2.22 public static final String HTTP_WHITEBOARD_RESOURCE_PREFIX = "osgi.http.whiteboard.resource.prefix"

Service property specifying the resource entry prefix for a resource service.

If a resource service is registered with this property, requests are served with bundle resources.

This prefix is used to map a requested resource to the bundle's entries. The value must not end with slash ("/") with the exception that a name of the form "/" is used to denote the root of the bundle. See the specification text for details on how HTTP requests are mapped.

The value of this service property must be of type String.

See Also HTTP_WHITEBOARD_RESOURCE_PATTERN

140.16.2.23 public static final String HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED = "osgi.http.whiteboard.servlet.asyncSupported"

Service property specifying whether a Servlet service supports asynchronous processing.

By default servlet services do not support asynchronous processing.

The value of this service property must be of type Boolean.

See Also Jakarta Servlet Specification Version 5.0, Asynchronous Processing [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#asynchronous-processing]

140.16.2.24 public static final String HTTP_WHITEBOARD_SERVLET_ERROR_PAGE = "osgi.http.whiteboard.servlet.errorPage"

Service property specifying whether a `Servlet` service acts as an error page.

The service property values may be the name of a fully qualified exception class, a three digit HTTP status code, the value "4xx" for all error codes in the 400 range, or the value "5xx" for all error codes in the 500 range. Any value that is not a three digit number, or one of the two special values is considered to be the name of a fully qualified exception class.

The value of this service property must be of type `String`, `String[]`, or `Collection<String>`.

140.16.2.25 public static final String HTTP_WHITEBOARD_SERVLET_INIT_PARAM_PREFIX = "servlet.init."

Service property prefix referencing a Servlet service.

For Servlet services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated servlet config. The prefix is removed from the service property name to build the initialization parameter name.

For Servlet services, the value of each initialization parameter service property must be of type `String`.

140.16.2.26 public static final String HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED = "osgi.http.whiteboard.servlet.multipart.enabled"

Service property specifying whether a `Servlet` service has enabled multipart request processing.

By default servlet services do not have multipart request processing enabled.

The value of this service property must be of type `Boolean`.

See Also Jakarta Servlet Specification Version 5.0, @MultipartConfig [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#_MultipartConfig]

140.16.2.27 public static final String HTTP_WHITEBOARD_SERVLET_MULTIPART_FILESIZETHRESHOLD = "osgi.http.whiteboard.servlet.multipart.fileSizeThreshold"

Service property specifying the size threshold after which the file will be written to disk.

When not set or when the value is not valid, the default threshold is determined by the implementation. This property is only evaluated if HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED is set to `true` .

The value of this service property must be of type `Integer`.

See Also Jakarta Servlet Specification Version 5.0, Deployment Descriptor Schema [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#deployment-descriptor-2]

140.16.2.28 public static final String HTTP_WHITEBOARD_SERVLET_MULTIPART_LOCATION = "osgi.http.whiteboard.servlet.multipart.location"

Service property specifying the location where the files can be stored on disk.

When not set the default location is defined by the value of the system property "java.io.tmpdir". This property is only evaluated if HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED is set to `true` .

The value of this service property must be of type `String`.

See Also Jakarta Servlet Specification Version 5.0, Deployment Descriptor Schema [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#deployment-descriptor-2]

140.16.2.29 public static final String HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXFILESIZE = "osgi.http.whiteboard.servlet.multipart.maxFileSize"

Service property specifying the maximum size of a file being uploaded.

When not set or when the value is not valid, the default maximum size is [@code -1} (no maximum size). This property is only evaluated if HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED is set to `true` .

The value of this service property must be of type `Long`.

See Also Jakarta Servlet Specification Version 5.0, Deployment Descriptor Schema [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#deployment-descriptor-2]

140.16.2.30 public static final String HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXREQUESTSIZE = "osgi.http.whiteboard.servlet.multipart.maxRequestSize"

Service property specifying the maximum request size.

When not set or when the value is not valid, the default maximum request size is `-1` (no maximum size). This property is only evaluated if HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED is set to `true` .

The value of this service property must be of type `Long`.

See Also Jakarta Servlet Specification Version 5.0, Deployment Descriptor Schema [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#deployment-descriptor-2]

140.16.2.31 public static final String HTTP_WHITEBOARD_SERVLET_NAME = "osgi.http.whiteboard.servlet.name"

Service property specifying the servlet name of a `Servlet` service.

The servlet is registered with this name and the name can be used as a reference to the servlet for filtering or request dispatching.

This name is in addition used as the value for the `ServletConfig.getServletName()` method. If this service property is not specified, the fully qualified name of the service object's class is used as the servlet name. Filter services may refer to servlets by this name in their HTTP_WHITEBOARD_FILTER_SERVLET service property to apply the filter to the servlet.

Servlet names should be unique among all servlet services associated with a single ServletContextHelper.

The value of this service property must be of type `String`.

140.16.2.32 public static final String HTTP_WHITEBOARD_SERVLET_PATTERN = "osgi.http.whiteboard.servlet.pattern"

Service property specifying the request mappings for a `Servlet` service.

The specified patterns are used to determine whether a request should be mapped to the servlet. Servlet services without this service property, HTTP_WHITEBOARD_SERVLET_ERROR_PAGE or HTTP_WHITEBOARD_SERVLET_NAME are ignored.

The value of this service property must be of type `String`, `String[]`, or `Collection<String>`.

See Also Jakarta Servlet Specification Version 5.0, Specification of Mappings [https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html#specification-of-mappings]

140.16.2.33 public static final String HTTP_WHITEBOARD_SPECIFICATION_VERSION = "2.0"

The version of the implementation capability for the Servlet Whiteboard specification

140.16.2.34 public static final String HTTP_WHITEBOARD_TARGET = "osgi.http.whiteboard.target"

Service property specifying the target filter to select the Servlet Whiteboard implementation to process the service.

An Servlet Whiteboard implementation can define any number of service properties which can be referenced by the target filter. The service properties should always include the osgi.http.endpoint service property if the endpoint information is known.

If this service property is not specified, then all Servlet Whiteboard implementations can process the service.

The value of this service property must be of type `String` and be a valid filter string.

140.16.3 public interface Preprocessor extends Filter

Services registered as a `Preprocessor` using a whiteboard pattern are executed for every request before the dispatching is performed.

If there are several services of this type, they are run in ranking order, the one with the highest ranking is used first.

The preprocessor is handled in the same way as filters. When a preprocessor is put into service Filter.init(jakarta.servlet.FilterConfig) is called, when it is not used anymore Filter.destroy() is called. As these preprocessors are run before dispatching and therefore the targeted servlet context is not known yet, jakarta.servlet.FilterConfig.getServletContext() returns the servlet context of the backing implementation. The same context is returned by the request object. The context path is the context path of this underlying servlet context. The passed in chain can be used to invoke the next preprocessor in the chain, or if the end of that chain is reached to start dispatching of the request. A preprocessor might decide to terminate the processing and directly generate a response.

Service properties with the prefix `HttpWhiteboardConstants#HTTP_WHITEBOARD_PREPROCESSOR_INIT_PARAM_PREFIX` are passed as init parameters to this service.

Concurrency Thread-safe

140.17 org.osgi.service.servlet.whiteboard.annotations

Servlet Whiteboard Annotations Package Version 2.0.

This package contains annotations that can be used to require the Servlet Whiteboard implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

140.17.1 Summary

- `RequireHttpWhiteboard` - This annotation can be used to require the Servlet Whiteboard implementation.

140.17.2 @RequireHttpWhiteboard

This annotation can be used to require the Servlet Whiteboard implementation. It can be used directly, or as a meta-annotation.

This annotation is applied to several of the Servlet Whiteboard component property annotations meaning that it does not normally need to be applied to Declarative Services components which use the Servlet Whiteboard.

Retention `CLASS`

Target `TYPE`, `PACKAGE`

140.18 org.osgi.service.servlet.whiteboard.propertytypes

Servlet Whiteboard Property Types Package Version 2.0.

When used as annotations, component property types are processed by tools to generate Component Descriptions which are used at runtime.

Bundles wishing to use this package at runtime must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.servlet.whiteboard.propertytypes; version="[2.0,3.0)"
```

140.18.1 Summary

- HttpWhiteboardContext - Component Property Type for the osgi.http.whiteboard.context.name and osgi.http.whiteboard.context.path service properties.
- HttpWhiteboardContextSelect - Component Property Type for the osgi.http.whiteboard.context.select service property.
- HttpWhiteboardFilterAsyncSupported - Component Property Type for the osgi.http.whiteboard.filter.asyncSupported service property.
- HttpWhiteboardFilterDispatcher - Component Property Type for the osgi.http.whiteboard.filter.dispatcher service property.
- HttpWhiteboardFilterName - Component Property Type for the osgi.http.whiteboard.filter.name service property.
- HttpWhiteboardFilterPattern - Component Property Type for the osgi.http.whiteboard.filter.pattern service property.
- HttpWhiteboardFilterRegex - Component Property Type for the osgi.http.whiteboard.filter.regex service property.
- HttpWhiteboardFilterServlet - Component Property Type for the osgi.http.whiteboard.filter.servlet service property.
- HttpWhiteboardListener - Component Property Type for the osgi.http.whiteboard.listener service property.
- HttpWhiteboardResource - Component Property Type for the osgi.http.whiteboard.resource.pattern and osgi.http.whiteboard.resource.prefix service properties.
- HttpWhiteboardServletAsyncSupported - Component Property Type for the osgi.http.whiteboard.servlet.asyncSupported service property.
- HttpWhiteboardServletErrorPage - Component Property Type for the osgi.http.whiteboard.servlet.errorPage service property.
- HttpWhiteboardServletMultipart - Component Property Type for the osgi.http.whiteboard.servlet.multipart.enabled, osgi.http.whiteboard.servlet.multipart.fileSizeThreshold, osgi.http.whiteboard.servlet.multipart.location, osgi.http.whiteboard.servlet.multipart.maxFileSize, and osgi.http.whiteboard.servlet.multipart.maxRequestSize service properties.
- HttpWhiteboardServletName - Component Property Type for the osgi.http.whiteboard.servlet.name service property.
- HttpWhiteboardServletPattern - Component Property Type for the osgi.http.whiteboard.servlet.pattern service property.
- HttpWhiteboardTarget - Component Property Type for the osgi.http.whiteboard.target service property.

140.18.2 @HttpWhiteboardContext

Component Property Type for the osgi.http.whiteboard.context.name and osgi.http.whiteboard.context.path service properties.

This annotation can be used on a ServletContextHelper to declare the values of the HTTP_WHITEBOARD_CONTEXT_NAME and HTTP_WHITEBOARD_CONTEXT_PATH service properties.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.2.1 String name

□ Service property identifying a servlet context helper name.

Returns The context name.

See Also HTTP_WHITEBOARD_CONTEXT_NAME

140.18.2.2 String path

□ Service property identifying a servlet context helper path.

Returns The context path.

See Also HTTP_WHITEBOARD_CONTEXT_PATH

140.18.2.3 String PREFIX_ = "osgi.http.whiteboard.context."

Prefix for the property name. This value is prepended to each property name.

140.18.3 @HttpWhiteboardContextSelect

Component Property Type for the osgi.http.whiteboard.context.select service property.

This annotation can be used on a Servlet Whiteboard component to declare the value of the HTTP_WHITEBOARD_CONTEXT_SELECT service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.3.1 String value

□ Service property identifying the select property of a Servlet Whiteboard component.

Returns The filter expression.

See Also HTTP_WHITEBOARD_CONTEXT_SELECT

140.18.3.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.4 @HttpWhiteboardFilterAsyncSupported

Component Property Type for the osgi.http.whiteboard.filter.asyncSupported service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.4.1 boolean asyncSupported default true

□ Service property identifying the asynchronous support of a filter.

Returns Whether the filter supports asynchronous processing.

See Also HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED

140.18.4.2 String PREFIX_ = "osgi.http.whiteboard.filter."

Prefix for the property name. This value is prepended to each property name.

140.18.5 @HttpWhiteboardFilterDispatcher

Component Property Type for the osgi.http.whiteboard.filter.dispatcher service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_DISPATCHER service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.5.1 DispatcherType[] value default jakarta.servlet.DispatcherType.REQUEST

□ Service property identifying dispatcher values for the filter.

Returns The dispatcher values for the filter.

See Also HTTP_WHITEBOARD_FILTER_DISPATCHER

140.18.5.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.6 @HttpWhiteboardFilterName

Component Property Type for the osgi.http.whiteboard.filter.name service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_NAME service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.6.1 String value

□ Service property identifying a filter name.

Returns The filter name.

See Also HTTP_WHITEBOARD_FILTER_NAME

140.18.6.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.7 @HttpWhiteboardFilterPattern

Component Property Type for the osgi.http.whiteboard.filter.pattern service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_PATTERN service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.7.1 String[] value

□ Service property identifying filter patterns.

Returns The filter patterns.

See Also HTTP_WHITEBOARD_FILTER_PATTERN

140.18.7.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.8 @HttpWhiteboardFilterRegex

Component Property Type for the osgi.http.whiteboard.filter.regex service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_REGEX service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.8.1 String[] value

□ Service property identifying filter regular expressions.

Returns The regular expressions for the filter.

See Also HTTP_WHITEBOARD_FILTER_REGEX

140.18.8.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.9 @HttpWhiteboardFilterServlet

Component Property Type for the osgi.http.whiteboard.filter.servlet service property.

This annotation can be used on a jakarta.servlet.Filter to declare the value of the HTTP_WHITEBOARD_FILTER_SERVLET service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.9.1 String[] value

□ Service property identifying the servlets for the filter.

Returns The servlet names.

See Also HTTP_WHITEBOARD_FILTER_SERVLET

140.18.9.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.10 @HttpWhiteboardListener

Component Property Type for the osgi.http.whiteboard.listener service property.

This annotation can be used on a Servlet Whiteboard listener to declare the value of the HTTP_WHITEBOARD_LISTENER service property as being Boolean.TRUE.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.10.1 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.11 @HttpWhiteboardResource

Component Property Type for the osgi.http.whiteboard.resource.pattern and osgi.http.whiteboard.resource.prefix service properties.

This annotation can be used on any service to declare the values of the HTTP_WHITEBOARD_RESOURCE_PATTERN and HTTP_WHITEBOARD_RESOURCE_PREFIX service properties.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.11.1 String[] pattern

□ Service property specifying the request mappings for resources. The specified patterns are used to determine whether a request should be mapped to resources.

Returns The resource patterns.

See Also HTTP_WHITEBOARD_RESOURCE_PATTERN

140.18.11.2 String prefix

□ Service property specifying the resource entry prefix for a resource service. This prefix is used to map a requested resource to the bundle's entries. The value must not end with slash ("/") with the exception that a name of the form "/" is used to denote the root of the bundle.

Returns The resource prefix.

See Also HTTP_WHITEBOARD_RESOURCE_PREFIX

140.18.11.3 String PREFIX_ = "osgi.http.whiteboard.resource."

Prefix for the property name. This value is prepended to each property name.

140.18.12 @HttpWhiteboardServletAsyncSupported

Component Property Type for the osgi.http.whiteboard.servlet.asyncSupported service property.

This annotation can be used on a jakarta.servlet.Servlet to declare the value of the HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.12.1 boolean asyncSupported default true

□ Service property identifying the asynchronous support of a servlet.

Returns Whether the servlet supports asynchronous processing.

See Also HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED

140.18.12.2 String PREFIX_ = "osgi.http.whiteboard.servlet."

Prefix for the property name. This value is prepended to each property name.

140.18.13 @HttpWhiteboardServletErrorPage

Component Property Type for the osgi.http.whiteboard.servlet.errorPage service property.

This annotation can be used on a jakarta.servlet.Servlet to declare the value of the HTTP_WHITEBOARD_SERVLET_ERROR_PAGE service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.13.1 String[] errorPage

□ Service property identifying the error pages of a servlet.

Returns The servlet error pages.

See Also HTTP_WHITEBOARD_SERVLET_ERROR_PAGE

140.18.13.2 String PREFIX_ = "osgi.http.whiteboard.servlet."

Prefix for the property name. This value is prepended to each property name.

140.18.14 @HttpWhiteboardServletMultipart

Component Property Type for the osgi.http.whiteboard.servlet.multipart.enabled, osgi.http.whiteboard.servlet.multipart.fileSizeThreshold, osgi.http.whiteboard.servlet.multipart.location, osgi.http.whiteboard.servlet.multipart.maxFileSize, and osgi.http.whiteboard.servlet.multipart.maxRequestSize service properties.

This annotation can be used on a jakarta.servlet.Servlet to declare the values of the HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED, HTTP_WHITEBOARD_SERVLET_MULTIPART_FILESIZETHRESHOLD, HTTP_WHITEBOARD_SERVLET_MULTIPART_LOCATION, HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXFILESIZE, and HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXREQUESTSIZE service properties.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.14.1 boolean enabled default true

□ Service property identifying the multipart handling of a servlet.

Returns Whether the servlet supports multipart handling.

See Also HTTP_WHITEBOARD_SERVLET_MULTIPART_ENABLED

140.18.14.2 int fileSizeThreshold default 0

□ Service property identifying the file size threshold for a multipart request handled by a servlet.

Returns The file size threshold for a multipart request..

See Also HTTP_WHITEBOARD_SERVLET_MULTIPART_FILESIZETHRESHOLD

140.18.14.3 String location default ""

□ Service property identifying the location for a multipart request handled by a servlet.

Returns The location for a multipart request..

See Also HTTP_WHITEBOARD_SERVLET_MULTIPART_LOCATION

140.18.14.4 long maxFileSize default -1L

□ Service property identifying the max file size for a multipart request handled by a servlet.

Returns The max file size for a multipart request..

See Also HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXFILESIZE

140.18.14.5 long maxRequestSize default -1L

□ Service property identifying the max request size for a multipart request handled by a servlet.

Returns The max request size for a multipart request..

See Also HTTP_WHITEBOARD_SERVLET_MULTIPART_MAXREQUESTSIZE

140.18.14.6 String PREFIX_ = "osgi.http.whiteboard.servlet.multipart."

Prefix for the property name. This value is prepended to each property name.

140.18.15 @HttpWhiteboardServletName

Component Property Type for the osgi.http.whiteboard.servlet.name service property.

This annotation can be used on a jakarta.servlet.Servlet to declare the value of the HTTP_WHITEBOARD_SERVLET_NAME service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.15.1 String value

□ Service property identifying a servlet name.

Returns The servlet name.

See Also HTTP_WHITEBOARD_SERVLET_NAME

140.18.15.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.16 @HttpWhiteboardServletPattern

Component Property Type for the osgi.http.whiteboard.servlet.pattern service property.

This annotation can be used on a jakarta.servlet.Servlet to declare the value of the HTTP_WHITEBOARD_SERVLET_PATTERN service property.

See Also Component Property Types

Retention CLASS

Target TYPE

140.18.16.1 String[] value

□ Service property identifying servlet patterns.

Returns The servlet patterns.

See Also HTTP_WHITEBOARD_SERVLET_PATTERN

140.18.16.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.18.17 @HttpWhiteboardTarget

Component Property Type for the `osgi.http.whiteboard.target` service property.

This annotation can be used on a Servlet Whiteboard service to declare the value of the HTTP_WHITEBOARD_TARGET service property.

See Also `Component Property Types`

Retention `CLASS`

Target `TYPE`

140.18.17.1 String value

□ Service property identifying the Servlet Whiteboard target.

Returns The Servlet Whiteboard target filter expression.

See Also HTTP_WHITEBOARD_TARGET

140.18.17.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

140.19 References

[1] *HTTP 1.0 Specification RFC-1945*
https://www.ietf.org/rfc/rfc1945.txt, May 1996

[2] *HTTP 1.1 Specifications RFCs 7230-7235*
https://tools.ietf.org/html/rfc7230
https://tools.ietf.org/html/rfc7231
https://tools.ietf.org/html/rfc7232
https://tools.ietf.org/html/rfc7233
https://tools.ietf.org/html/rfc7234
https://tools.ietf.org/html/rfc7235

[3] *HTTP/2 Specifications*
https://http2.github.io

[4] *Jakarta Servlet 5.0 Specification*
https://jakarta.ee/specifications/servlet/5.0/

[5] *Portable Java Contract Definitions*
https://docs.osgi.org/reference/portable-java-contracts.html

[6] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*
https://www.ietf.org/rfc/rfc2617.txt

[7] *Whiteboard Pattern*
https://docs.osgi.org/whitepaper/whiteboard-pattern/

[8] *Core Service Hooks*
OSGi Core, Chapter 55 Service Hook Service Specification

[9] *Http Whiteboard Specification*
https://docs.osgi.org/specification/osgi.cmpn/8.0.0/service.http.whiteboard.html

[10] *Http Service Specification*
https://docs.osgi.org/specification/osgi.cmpn/8.0.0/service.http.html

140.20 Changes

- This specification is providing the same functionality as its predecessor, [9] *Http Whiteboard Specification*, with the switch from the Java Servlet API to the Jakarta Servlet API.
- Removed the reference to the [10] *Http Service Specification* specying the behaviour if an implementation is also implementing that specification.

141 Device Abstraction Layer Specification

Version 1.0

141.1 Introduction

The Internet-of-Things (IoT) has a major impact in the IT industry. It requires backend systems to receive information from sensors, actuators, and appliances in various vertical markets such as Smart Home, eHealth, industrial automation, logistics, and automotive telematics. Application developers have to face the still increasing amount of communication protocols which are the major hurdle for interoperability.

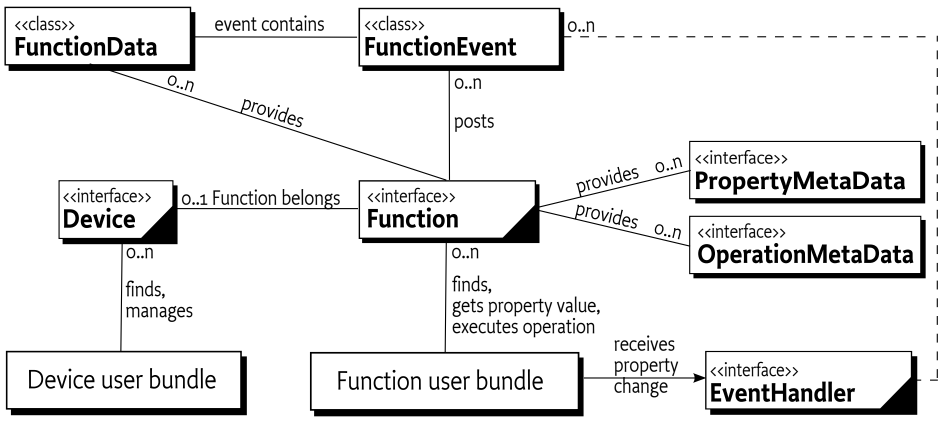
The Device Abstraction Layer specification provides a unified interface for application developers to interact with sensor, devices, etc. connected to a gateway. Application developers don't have to deal with protocol specific details which simplifies the development of their applications.

The remote device control provides an opportunity to save energy, to support better security, to save your time during daily tasks and more. The devices can play different roles in their networks as event reporters, controllers, etc. That dynamic behavior is well mappable to the dynamic OSGi service registry. When a new device is available in the network, there is a registration of a Device service. It realizes basic set of management operations and provides a rich set of properties. The applications can track the device status, read descriptive information and follow the device relations. A set of functions can belong to a single device. They represent the device operations and related properties in an atomic way. The device functions can be found in the OSGi service registry. The applications are allowed to get directly the required functions if they don't need information about the device. For example, light device is registered as Device service and there is Function service to turn on and turn off the light. The application can operate with the light control service without access to the device service.

141.1.1 Entities

- *Device* - represents the device in the OSGi service registry. It's described with a set of service properties and provides basic management operations.
- *Function* - atomic functional entity like switch or sensor. The function can belong to a device. The function provides a set of properties and operations.
- *FunctionEvent* - asynchronous event. It's posted through EventAdmin service and notifies for Function property change.
- *FunctionData* - data structure which carries Function property value with extra metadata.
- *PropertyMetadata* and *OperationMetadata* - contain metadata about the Function properties and operations.

Figure 141.1 Device Abstraction Layer Overview



141.2 Device Category

The device category defined in the scope of the Device Access service specification is called DAL. DEVICE_CATEGORY constant contains the category name.

141.3 Device Service

The Device interface is dedicated to a common access to the devices provided by different protocols. It can be mapped one to one with the physical device, but can be mapped only with a given functional part of the device. Another mapping can be a device realized with a set of Device services and different relations between them. Device service can represent pure software unit. For example, it can simulate the real device work. There are basic management operations for removal and property access. New protocol devices can be supported with the registration of new Device services.

If the underlying protocol and the implementation allow, the Device services must be registered again after the OSGi framework restarts. The service properties must be restored, the supported functions must be registered and Device relations must be visible to the applications.

141.3.1 Device Service Properties

The OSGi service registry has the advantage of being easily accessible. The services can be filtered and accessed with their properties. The Device service has a rich set of such properties:

- SERVICE_UID – Specifies the device unique identifier. It's a mandatory property. The value type is java.lang.String. To simplify the unique identifier generation, the property value must follow the rule:

  ```
  UID ::= driver-name ':' device-id
  ```

  - UID – device unique identifier
  - driver-name – the value of the Device.SERVICE_DRIVER service property
  - device-id – device unique identifier in the scope of the driver
- SERVICE_REFERENCE_UIDS – Specifies the reference device unique identifiers. It's an optional property. The value type is java.lang.String[]. It can be used to represent different relationships between the devices. For example, The EnOcean controller can have a reference to the USB dongle.

- SERVICE_DRIVER – Specifies the device driver name. For example, EnOcean, Z-Wave, Bluetooth, etc. It's a mandatory property. The value type is java.lang.String.
- SERVICE_NAME – Specifies the device name. It's an optional property. The value type is java.lang.String.
- SERVICE_STATUS – Specifies the current device status. It's a mandatory property. The value type java.lang.Integer. The possible values are:
  - STATUS_REMOVED – Indicates that the device has been removed from the network. That status must be set as the last device status and after that the device service can be unregistered from the service registry. The status is available for stale device services too. All transitions to this status are described in *Removed* on page 847.
  - STATUS_OFFLINE – Indicates that the device is currently not available for operations. The end device is available in the network and can become online later. The controller is unplugged or there is no connection. All transitions to and from this status are described in detail in *Offline* on page 847.
  - STATUS_ONLINE – Indicates that the device is currently available for operations. The recent communication with the device has been passed through. All transitions to and from this status are described in detail in *Online* on page 848.
  - STATUS_PROCESSING – Indicates that the device is currently busy with an operation. All transitions to and from this status are described in detail in *Processing* on page 849.
  - STATUS_NOT_INITIALIZED – Indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. All transitions to and from this status are described in detail in *Not Initialized* on page 850.
  - STATUS_NOT_CONFIGURED – Indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. All transitions to and from this status are described in detail in *Not Configured* on page 851.
- SERVICE_STATUS_DETAIL – Provides the reason for the current device status. It's an optional property. The property value cannot be externally set or modified. The value type is java.lang.Integer. There are two value categories. Positive values indicate the reason for the current status like STATUS_DETAIL_CONNECTING. Negative values indicate errors related to the current device status like STATUS_DETAIL_BROKEN. The list with defined status details is:
  - STATUS_DETAIL_CONNECTING – The device is currently connecting to the network. The status detail indicates the reason with a positive value 1. The device status must be STATUS_PROCESSING.
  - STATUS_DETAIL_INITIALIZING – The device is currently in process of initialization. The status detail indicates the reason with a positive value 2. The network controller initializing means that information about the network is currently read. The device status must be STATUS_PROCESSING.
  - STATUS_DETAIL_REMOVING – The device is leaving the network. The status detail indicates the reason with positive value 3. The device status must be STATUS_PROCESSING.
  - STATUS_DETAIL_FIRMWARE_UPDATING – The device firmware is updating. The status detail indicates the reason with positive value 4. The device status must be STATUS_PROCESSING.
  - STATUS_DETAIL_CONFIGURATION_UNAPPLIED – The device configuration is not applied. The status detail indicates an error with a negative value -1. The device status must be STATUS_NOT_CONFIGURED.
  - STATUS_DETAIL_BROKEN – The device is broken. The status detail indicates an error with a negative value -2. The device status must be STATUS_OFFLINE.
  - STATUS_DETAIL_COMMUNICATION_ERROR – The device communication is problematic. The status detail indicates an error with a negative value -3. The device status must be STATUS_ONLINE or STATUS_NOT_INITIALIZED.

- STATUS_DETAIL_DATA_INSUFFICIENT – The device doesn't provide enough information and cannot be determined. The status detail indicates an error with a negative value -4. The device status must be STATUS_NOT_INITIALIZED.
- STATUS_DETAIL_INACCESSIBLE – The device is not accessible and further communication is not possible. The status detail indicates an error with a negative value -5. The device status must be STATUS_OFFLINE.
- STATUS_DETAIL_CONFIGURATION_ERROR – The device cannot be configured. The status detail indicates an error with a negative value -6. The device status must be STATUS_NOT_CONFIGURED.
- STATUS_DETAIL_DUTY_CYCLE – The device is in duty cycle. The status detail indicates an error with a negative value -7. The device status must be STATUS_OFFLINE.

Custom status details are allowed, but they must not overlap the specified codes. To prevent possible collisions with further updates, custom codes must be greater than 100 and less than -100. Table 141.1 contains the mapping of the status details to the statuses.

Table 141.1 Status detail to status mapping.

Status Detail	Status
CONNECTING	PROCESSING
INITIALIZING	PROCESSING
REMOVING	PROCESSING
FIRMWARE_UPDATING	PROCESSING
CONFIGURATION_UNAPPLIED	NOT_CONFIGURED
BROKEN	OFFLINE
COMMUNICATION_ERROR	ONLINE, NOT_INITIALIZED
DATA_INSUFFICIENT	NOT_INITIALIZED
INACCESSIBLE	OFFLINE
CONFIGURATION_ERROR	NOT_CONFIGURED
DUTY_CYCLE	OFFLINE

- SERVICE_HARDWARE_VENDOR – Specifies the device hardware vendor. It's an optional property. The value type is java.lang.String.
- SERVICE_HARDWARE_VERSION – Specifies the device hardware version. It's an optional property. The value type is java.lang.String.
- SERVICE_FIRMWARE_VENDOR – Specifies the device firmware vendor. It's an optional property. The value type is java.lang.String.
- SERVICE_FIRMWARE_VERSION – Specifies the device firmware version. It's an optional property. The value type is java.lang.String.
- SERVICE_TYPES – Specifies the device types. It's an optional property. The value type is java.lang.String[].
- SERVICE_MODEL – Specifies the device model. It's an optional property. The value type is java.lang.String.
- SERVICE_SERIAL_NUMBER – Specifies the device serial number. It's an optional property. The value type is java.lang.String.

The next code snippet prints all online devices.

```
ServiceReference[] deviceSRefs = context.getServiceReferences(
    Device.class.getName(),
    '(' + Device.SERVICE_STATUS + '=' + Device.STATUS_ONLINE + ')');
if (deviceSRefs != null) {
    for (int i = 0; i < deviceSRefs.length; i++) {
        printDevice(deviceSRefs[i]);
    }
}
```

Applications need to have an access to the device properties. For convenience, there are helper methods:

- getServiceProperty(String) – Returns the current value of the specified property. The method will return the same value as org.osgi.framework.ServiceReference.getProperty(String) for the service reference of this device.
- getServicePropertyKeys() – Returns an array with all device service property keys. The method will return the same value as org.osgi.framework.ServiceReference.getPropertyKeys() for the service reference of this device.

141.3.2 Device Registration

The devices are registered as services in the OSGi service registry. The service interface is org.osgi.service.dal.Device. There is a registration order. Device services are registered last on start up. Before their registration, there is Function service registration. The function registration procedure is described in *Function Registration* on page 853.

The OSGi service registry provides an access to the services, but there are no management operations like remove a given service. The service provider is responsible to register and unregister own services. That design doesn't provide an option to remove the device services. The Device interface fills this gap with remove() method. It's a callback to the service provider to remove the device from the network. The method can be optionally implemented. java.lang.UnsupportedOperationException can be thrown if the method is not supported. When the remove() is called:

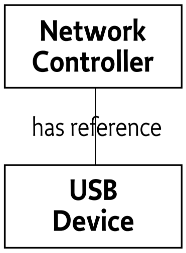
- An appropriate command will be synchronously send to the device. As a result it can leave the network.
- The device status will be set to STATUS_REMOVED.
- The related device service will be unregistered from the OSGi service registry.

There is an unregistration order. The registration reverse order is used when the services are unregistered. Device services are unregistered first before Function services.

141.3.3 Reference Devices

Device service can have a reference to other devices. That link can be used to represent different relationships between devices. For example, the EnOcean dongle can be used as USB Device and EnOcean network controller Device. The network controller device can have a reference to the physical USB device as it's depicted on the next diagram.

Figure 141.2 Device Reference



The related service property is SERVICE_REFERENCE_UIDS.

141.3.4 Device Status Transitions

The device status reveals the device availability. It can demonstrate that device is currently not available for operations or that the device requires some additional configuration steps. The status can move between the different values according to the rules defined in this section. The status transitions are summarized in Table 141.2, visualized on Figure 141.3 and described in detail in the next sections. The initial device status is always STATUS_PROCESSING. When device info is processed, the device can go to another status. The last possible device status is STATUS_REMOVED. The status must be set when the device is removed from the network. After that status, the device service will be unregistered.

Figure 141.3 Device Status Transitions

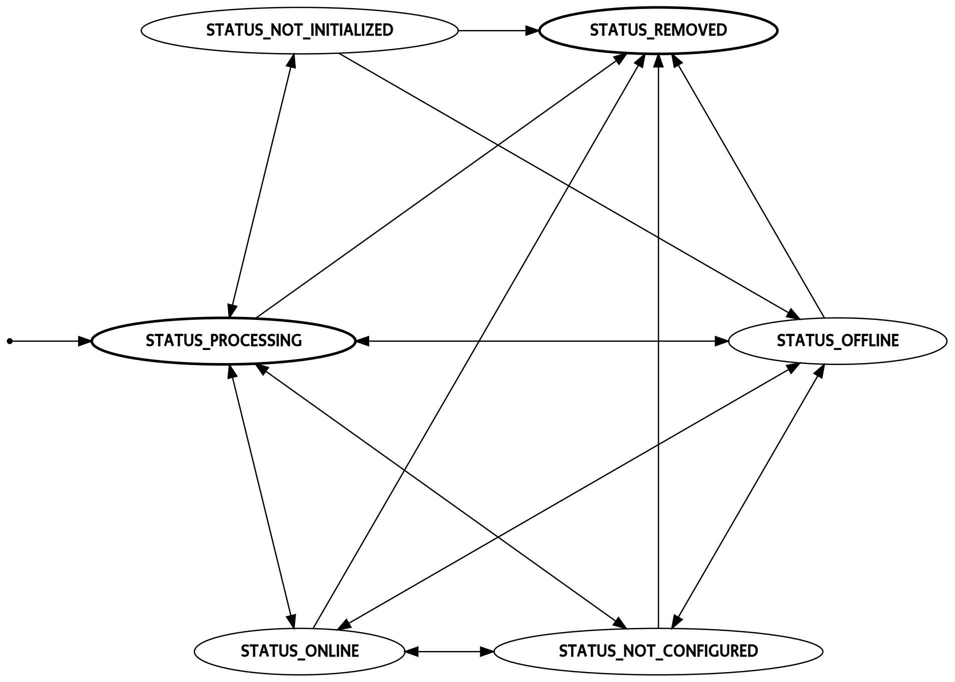


Table 141.2 Device Status Transitions

From\To Status	PROCESSING	ONLINE	OFFLINE	NOT INITIALIZED	NOT CONFIGURED	REMOVED
PROCESSING	-	Initial device data has been read.	Device is not accessible.	Initial device data has been partially read.	Device has a pending configuration.	Device has been removed.
ONLINE	Device data is processing.	-	Device is not accessible.	-	Device has a new pending configuration.	Device has been removed.
OFFLINE	Device data is processing.	Device data has been read.	-	-	Device has a pending configuration.	Device has been removed.
NOT INITIALIZED	Device data is processing.	-	Device is not accessible.	-	-	Device has been removed.

From\To Status	PROCESSING	ONLINE	OFFLINE	NOT INITIALIZED	NOT CONFIGURED	REMOVED
NOT CONFIGURED	Device data is processing.	Device pending configuration is satisfied.	Device is not accessible.	-	-	Device has been removed.
REMOVED	-	-	-	-	-	-

141.3.4.1 Removed

The device can go to STATUS_REMOVED from any other status. Once reached, the device status cannot be updated any more. The device has been removed from the network and the device service is unregistered from the OSGi service registry. If there are stale references to the Device service, their status will be set to STATUS_REMOVED.

The common way for a given device to be removed is remove() method. When the method returns, the device status will be STATUS_REMOVED. It requires a synchronous execution of the operation.

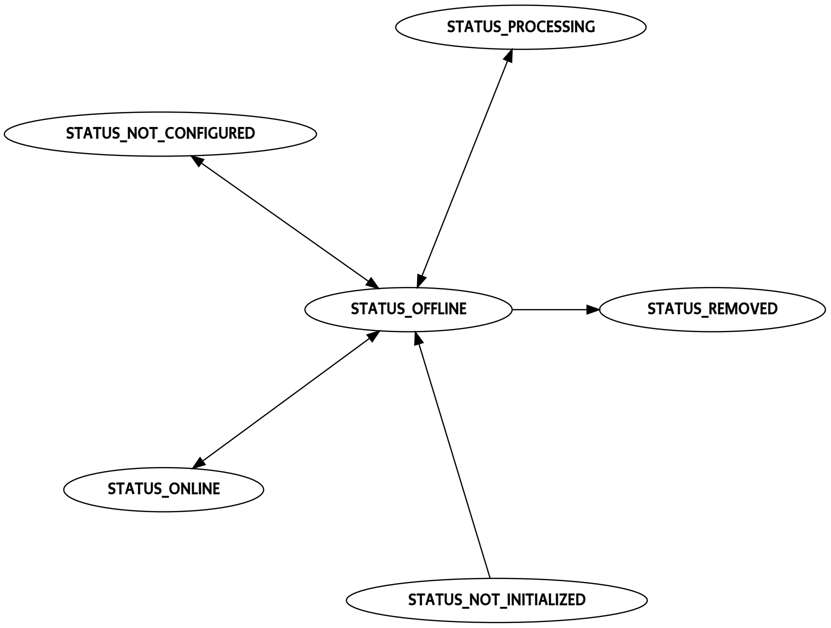
141.3.4.2 Offline

The STATUS_OFFLINE indicates that the device is currently not available for operations. That status can be set, because of different reasons. The network controller has been unplugged, the connection to the device has been lost, etc. The device can move to this status from any other status with the exception of STATUS_REMOVED. Transitions to and from this status are:

- From STATUS_OFFLINE to STATUS_REMOVED – The device has been removed. The status can be set as a result of remove() method call.
- From STATUS_OFFLINE to STATUS_PROCESSING – Device data is processing.
- From STATUS_OFFLINE to STATUS_NOT_CONFIGURED – The device has a pending configuration.
- From STATUS_OFFLINE to STATUS_ONLINE – Device data has been read and the device is currently available for operations.
- From STATUS_OFFLINE to STATUS_NOT_INITIALIZED – That transition is not possible, because the status have to go through STATUS_PROCESSING. If the processing is unsuccessful, STATUS_NOT_INITIALIZED will be set.
- To STATUS_OFFLINE from STATUS_REMOVED – That transition is not possible. If the device has been removed, the service will be unregistered from the service registry.
- To STATUS_OFFLINE from STATUS_PROCESSING – The device is not accessible any more while device data is processing.
- To STATUS_OFFLINE from STATUS_NOT_CONFIGURED – The device with pending configuration is not accessible any more.
- To STATUS_OFFLINE from STATUS_ONLINE – The online device is not accessible any more.
- To STATUS_OFFLINE from STATUS_NOT_INITIALIZED – The not initialized device is not accessible any more.

The possible transitions are summarized on Figure 141.4.

Figure 141.4 Transitions to and from STATUS_OFFLINE



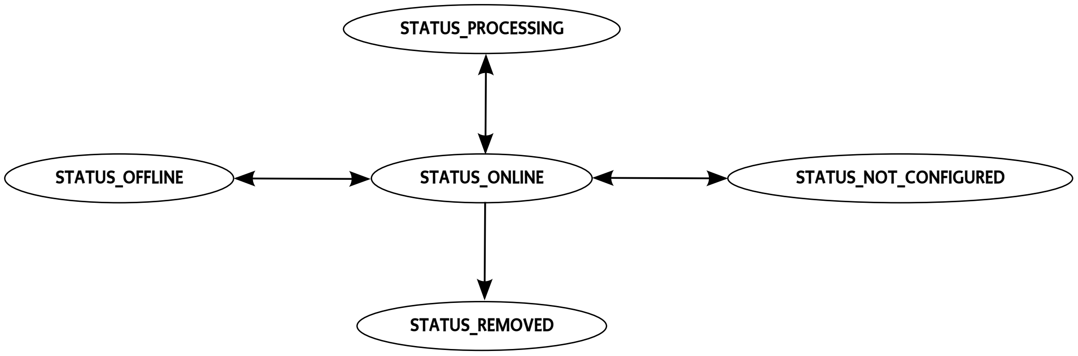
141.3.4.3 Online

The STATUS_ONLINE indicates that the device is currently available for operations. The online devices are initialized and ready for use. Transitions to and from this status are:

- From STATUS_ONLINE to STATUS_REMOVED – The device has been removed. The status can be set as a result of remove() method call.
- From STATUS_ONLINE to STATUS_PROCESSING – The device data is processing.
- From STATUS_ONLINE to STATUS_NOT_CONFIGURED – The device has a pending configuration.
- From STATUS_ONLINE to STATUS_OFFLINE – The online device is not accessible any more.
- From STATUS_ONLINE to STATUS_NOT_INITIALIZED – That transition is not possible. Online devices are initialized.
- To STATUS_ONLINE from STATUS_REMOVED – That transition is not possible. If the device has been removed, the service will be unregistered from the service registry.
- To STATUS_ONLINE from STATUS_PROCESSING – Initial device data has been read. The device is available for operations.
- To STATUS_ONLINE from STATUS_NOT_CONFIGURED – The device pending configuration is satisfied.
- To STATUS_ONLINE from STATUS_OFFLINE – The device is accessible for operations.
- To STATUS_ONLINE from STATUS_NOT_INITIALIZED – That transition is not possible. The device data has to be processed and then the device can become online. Intermediate status STATUS_PROCESSING will be used.

The possible transitions are summarized on Figure 141.5.

Figure 141.5 Transitions to and from STATUS_ONLINE



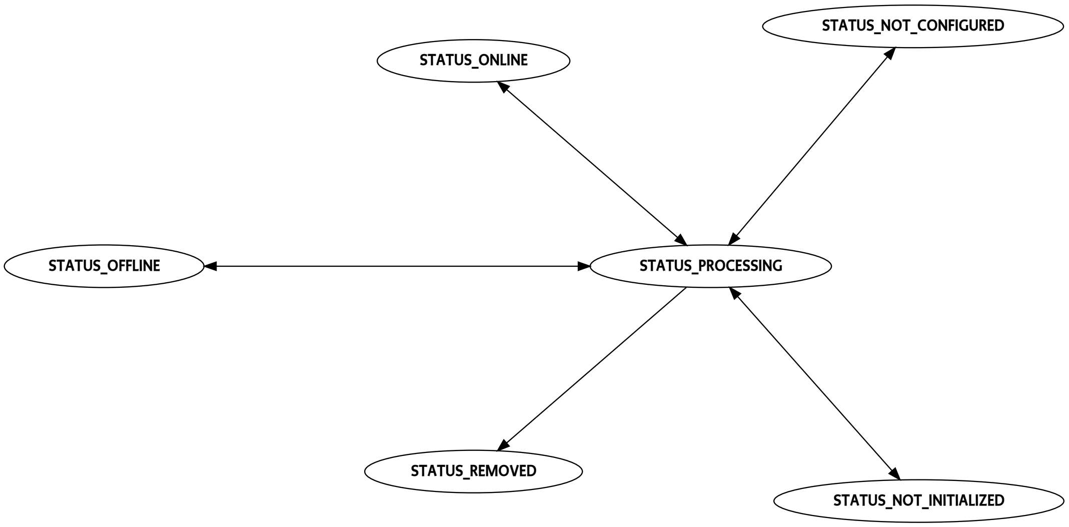
141.3.4.4 Processing

The status indicates that the device is currently busy with an operation. It can be time consuming operation and can result to any other status. The operation processing can be reached by any other status except STATUS_REMOVED. For example, offline device requires some data processing to become online. It will apply this status sequence: STATUS_OFFLINE, STATUS_PROCESSING and STATUS_ONLINE. Transitions to and from this status are:

- From STATUS_PROCESSING to STATUS_REMOVED – The device has been removed. The status can be set as a result of remove() method call.
- From STATUS_PROCESSING to STATUS_ONLINE – Initial device data has been read. The device is available for operations.
- From STATUS_PROCESSING to STATUS_NOT_CONFIGURED – The device has a pending configuration.
- From STATUS_PROCESSING to STATUS_OFFLINE – The device is not accessible any more.
- From STATUS_PROCESSING to STATUS_NOT_INITIALIZED – The device initial data is partially read.
- To STATUS_PROCESSING from STATUS_REMOVED – That transition is not possible. If the device has been removed, the service will be unregistered from the service registry.
- To STATUS_PROCESSING from STATUS_ONLINE – The device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_CONFIGURED – The device pending configuration is satisfied and the device is busy with an operation.
- To STATUS_PROCESSING from STATUS_OFFLINE – The device is busy with an operation.
- To STATUS_PROCESSING from STATUS_NOT_INITIALIZED – The device initial data is processing.

The possible transitions are summarized on Figure 141.6.

Figure 141.6 Transitions to and from STATUS_PROCESSING



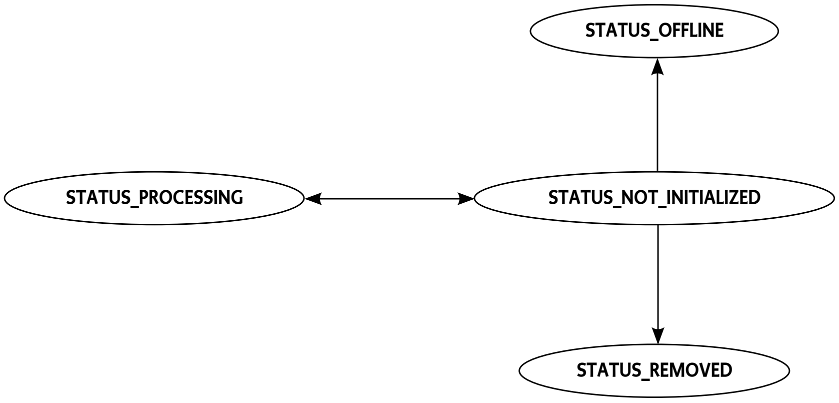
141.3.4.5 Not Initialized

The status indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. The not initialized device requires some data processing to become online. STATUS_PROCESSING is used as an intermediate status. Transitions to and from this status are:

- From STATUS_NOT_INITIALIZED to STATUS_REMOVED – The device has been removed. The status can be set as a result of remove() method call.
- From STATUS_NOT_INITIALIZED to STATUS_PROCESSING – The device data is processing.
- From STATUS_NOT_INITIALIZED to STATUS_NOT_CONFIGURED – That transition is not possible. Device requires some data processing.
- From STATUS_NOT_INITIALIZED to STATUS_OFFLINE – The device is not accessible any more.
- From STATUS_NOT_INITIALIZED to STATUS_ONLINE – That transition is not possible. Device requires some data processing to become online.
- To STATUS_NOT_INITIALIZED from STATUS_REMOVED – That transition is not possible. If the device has been removed, the service will be unregistered from the service registry.
- To STATUS_NOT_INITIALIZED from STATUS_PROCESSING – Device data is partially read.
- To STATUS_NOT_INITIALIZED from STATUS_NOT_CONFIGURED – That transition is not possible. When device pending configuration is satisfied, the device requires additional data processing.
- To STATUS_NOT_INITIALIZED from STATUS_OFFLINE – That transition is not possible. Device requires some data processing and then can become not initialized.
- To STATUS_NOT_INITIALIZED from STATUS_ONLINE – That transition is not possible. The online device is initialized.

The possible transitions are summarized on Figure 141.7.

Figure 141.7 Transitions to and from STATUS_NOT_INITIALIZED



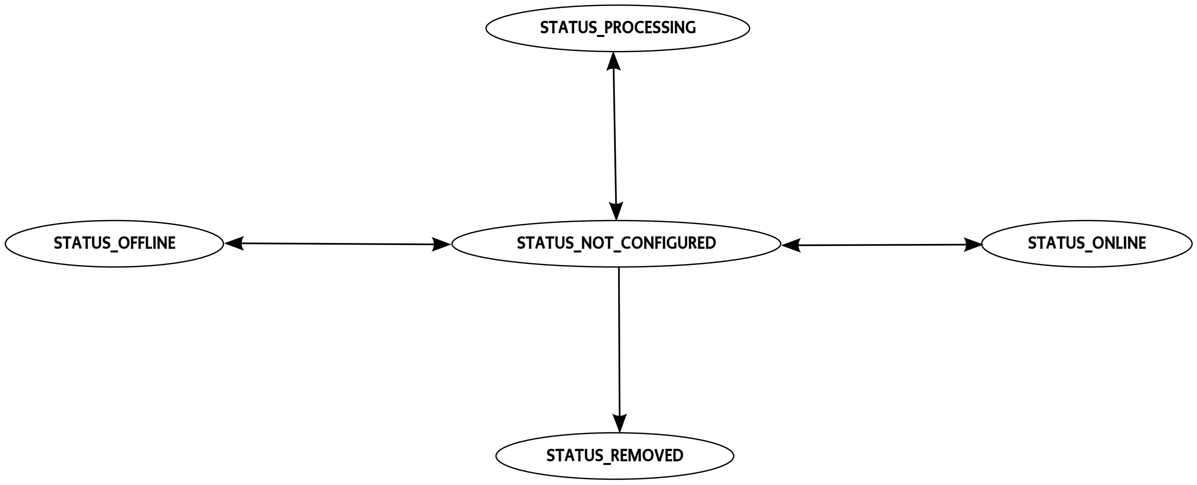
141.3.4.6 Not Configured

Indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. For example, a given device button has to be pushed. That status doesn't have transitions with STATUS_NOT_INITIALIZED, because some data processing is required. Transitions to and from this status are:

- From STATUS_NOT_CONFIGURED to STATUS_REMOVED – The device has been removed. The status can be set as a result of remove() method call.
- From STATUS_NOT_CONFIGURED to STATUS_PROCESSING – The device pending configuration is satisfied and some additional data processing is required.
- From STATUS_NOT_CONFIGURED to STATUS_ONLINE – The device pending configuration is satisfied.
- From STATUS_NOT_CONFIGURED to STATUS_OFFLINE – The device is not accessible any more.
- From STATUS_NOT_CONFIGURED to STATUS_NOT_INITIALIZED – That transition is not possible. When device pending configuration is satisfied, the device requires additional data processing.
- To STATUS_NOT_CONFIGURED from STATUS_REMOVED – That transition is not possible. If the device has been removed, the service will be unregistered from the service registry.
- To STATUS_NOT_CONFIGURED from STATUS_PROCESSING – Initial device data has been read but there is a pending configuration.
- To STATUS_NOT_CONFIGURED from STATUS_ONLINE – The device has a pending configuration.
- To STATUS_NOT_CONFIGURED from STATUS_OFFLINE – The device is going to be online, but has a pending configuration.
- To STATUS_NOT_CONFIGURED from STATUS_NOT_INITIALIZED – That transition is not possible. Device requires some data processing.

The possible transitions are summarized on Figure 141.8.

Figure 141.8 Transitions to and from STATUS_NOT_CONFIGURED



141.4 Function Service

The user applications have full control over the device with the Function services. Synchronous or asynchronous operations can trigger different actions. For example, turn on or off the light, can change the room temperature, send an user notification, etc. The action result can be reported immediately or later in case of concurrent execution. As a result, a Function property can be updated. The property is the device value container. It can provide, sensor information, meter data, the switch current position, etc. Different property access types allow the applications to read, write or receive events.

141.4.1 Function Service Properties

The OSGi service registry has the advantage of being easily accessible. The services can be filtered and accessed with their properties. The function service has a rich set of such properties:

- SERVICE_UID – mandatory service property. The property value is the function unique identifier. The value type is java.lang.String. To simplify the unique identifier generation, the property value must follow the rule:

  ```
  function UID ::= device-id ':' function-id
  ```

  - function UID – function unique identifier
  - device-id – the value of the Device.SERVICE_UID Device service property
  - function-id – function identifier in the scope of the device

  If the function is not bound to a device, the function unique identifier can be device independent.
- SERVICE_TYPE – optional service property. The service property value contains the function type. For example, the sensor function can have different types like temperature, pressure, etc. The value type is java.lang.String.

  Organizations that want to use function types that do not clash with OSGi Working Group defined types should prefix their types in own namespace.
- SERVICE_VERSION – optional service property. The service property value contains the function version. That version can point to specific implementation version and vary in the different vendor implementations. The value type is java.lang.String.

- SERVICE_DEVICE_UID – optional service property. The property value is the device identifier. The function belongs to this device. The value type is java.lang.String.
- SERVICE_REFERENCE_UIDS – optional service property. The service property value contains the reference function unique identifiers. The value type is java.lang.String[]. It can be used to represent different relationships between the functions.
- SERVICE_DESCRIPTION – optional service property. The property value is the function description. The value type is java.lang.String.
- SERVICE_OPERATION_NAMES – optional service property. The property is missing when there are no function operations and property must be set when there are function operations. The property value is the function operation names. The value type is java.lang.String[]. It's not possible to exist two or more function operations with the same name i.e. the operation overloading is not allowed.
- SERVICE_PROPERTY_NAMES – optional service property. The property is missing when there are no function properties and property must be set when there are function properties. The property value is the function property names. The value type is java.lang.String[]. It's not possible to exist two or more function properties with the same name.

141.4.2 Function Registration

On start up, the Function services are registered before the Device service. It's possible that SERVICE_DEVICE_UID points to missing service at the moment of the registration. The reverse order is used when the services are unregistered. Device service is unregistered before the Function services. The device registration procedure is available in *Device Registration* on page 845.

The Function service should be registered only under the function class hierarchy. Other classes can be used if there are no ambiguous representations. For example, an ambiguous representation can be a function registered under two independent function classes like BinarySwitch and Meter. In this example, both functions support the same property “state” with different meaning. getPropertyMetadata(String propertyName) method cannot determinate which property is requested. It can be BinarySwitch “state” or Meter “state”.

To simplify the generic function discovery, the Function interface must be used for the service registration. In this way, the generic applications can easily find all services, which are functions in the service registry. Because of this rule, this registration is not allowed:

```
context.registerService(MeterV1.class.getName(), this, regProps);
```

If the implementation would like to mark that there is a function, but no specific function interface exists, the registration can be:

```
context.registerService(Function.class.getName(), this, regProps);
```

Note that such functions usually don't have operations and properties.

141.4.3 Function Interface

Function is built by a set of properties and operations. The function can have unique identifier, type, version, description, link to the Device service and information about the referenced functions. Function interface must be the base interface for all functions. If the device provider defines custom functions, all of them must extend Function interface. It provides a common access to the operations and properties metadata.

There are some general type rules, which unify the access to the function data. They make easier the transfer over different protocols. All properties and operation arguments must use one of:

- Java primitive type or corresponding reference type.
- Numerical type i.e. the type which extends java.lang.Number. The numerical type must follow these conventions:

  - The type must provide a public static method called valueOf that returns an instance of the given type and takes a single String argument or a public constructor which takes a single String argument.
  - The String argument from the previous bullet can be provided by toString() method of the instance.
- java.lang.String
- Java Bean, but its properties must use those rules. Java Bean is defined in [1] *JavaBeans Spec*.
- java.util.Map instance. The map keys can be java.lang.String. The values of a single type follow these rules.
- Array of defined types.

In order to provide common behavior, all functions must follow a set of common rules related to the implementation of their setters, getters, operations and events:

- The setter method must be executed synchronously. If the underlying protocol can return response to the setter call, it must be awaited. It simplifies the property value modification and doesn't require asynchronous callback.
- The operation method must be executed synchronously. If the underlying protocol can return an operation confirmation or response, it must be awaited. It simplifies the operation execution and doesn't require asynchronous callback.
- The getter must return the last know cached property value. The device implementation is responsible to keep that value up to date. It'll speed up the applications when the function property values are collected. The same cached value can be shared between a few requests instead of a few calls to the real device.
- The function operations, getters and setters must not override java.lang.Object and this interface methods. For example:
  - hashCode() – it's java.lang.Object method and invalid function operation;
  - wait() – it's java.lang.Object method and invalid function operation;
  - getClass() – it's java.lang.Object method and invalid function getter;
  - getPropertyMetadata(String propertyName) – it's org.osgi.service.dal.Function method and invalid function getter.

141.4.4 Function Operations

Function operations are the main callable units. They can perform a specific task on the device like turn on or turn off. They can be used by the applications to control the device. Operation names are available as a value of the service property SERVICE_OPERATION_NAMES. The operations are identified by their names. It's not possible to exist two operations with the same name i.e. overloaded operations are not allowed. They cannot override the property accessor methods. The operations are regular java methods. That implies that they have zero or more arguments and zero or one return value. The operation arguments and return value must follow the general type rules.

The operations can be optionally described with metadata. Metadata is accessible with getOperationMetadata(String) method. The result provides metadata about the operation, operation arguments and result value. Operation arguments and result value are using the same metadata as the function properties. The full details are defined in the next section.

141.4.5 Function Properties

Function properties are class fields. Their values can be read with getter methods and can be set with setter methods. The property names are available as a value of the service property SERVICE_PROPERTY_NAMES. The properties are identified by their names. It's not possible to exist two properties with the same name.

The function properties must be integrated according to these rules:

- Getter methods must be available for all properties with ACCESS_READABLE access.
- Getter method must return a subclass of FunctionData.
- Setter methods must be available for all properties with ACCESS_WRITABLE access.
- Setter methods can be any combination of:
  - Setter method which accepts a subclass of FunctionData.
  - Setter method which accepts the values used by the FunctionData subclass, if there are no equal types.

  It's possible to have only one or both of them. Examples:
  - There is MyFunctionData bean with BigDecimal value for a data property. Valid setters are setData(MyFunctionData data) and setData(BigDecimal data).
  - There is MySecondFunctionData bean with BigDecimal prefix and BigDecimal suffix for a data property. The prefix and suffix are using equal types and we cannot have a setter with the values used by MySecondFunctionData. The only one possible setter is setData(MySecondFunctionData data).
- No methods are required for properties with ACCESS_EVENTABLE access.

The accessor method names must be defined according to [1] *JavaBeans Spec*.

The properties can be optionally described with a set of metadata properties. The property values can be collected with getPropertyMetadata(String) method. The method result is PropertyMetadata with:

- Minimum value – available through getMinValue(String). The minimum value can be different for the different units.
- Maximum value – available through getMaxValue(String). The maximum value can be different for the different units.
- Enumeration of values – available through getEnumValues(String). The array of the possible values is sorted in increasing order according to the given unit.
- Step – available through getStep(String). The difference between two values in series. For example, if the range is [0, 100], the step can be 10.
- Property access – available as a value in getMetadata(String) result map. It's a bitmap of java.lang.Integer type and doesn't depend on the given unit. The access is available only for the function properties and it's missing for the operation arguments and result metadata. The bitmap can be any combination of:
  - ACCESS_READABLE – Marks the property as a readable. Function must provide a getter method for this property according to [1] *JavaBeans Spec*. Function operations must not be overridden by this getter method.
  - ACCESS_WRITABLE – Marks the property as writable. Function must provide a setter method for this property according to [1] *JavaBeans Spec*. Function operations must not be overridden by this setter method.
  - ACCESS_EVENTABLE – Marks the property as eventable. Function must not provide special methods because of this access type. FunctionEvent is sent on property change. Note that the event can be sent when there is no value change.
- Units - available as a value in getMetadata(String) result map. They can be requested with key UNITS. The value contains the property supported units. The property value type is java.lang.String[]. The array first element at index 0 represents the default unit. Each unit must follow those rules:
  - The International System of Units must be used where it's applicable. For example, kg for kilogram and km for kilometer.
  - If the unit name matches to a Unicode symbol name, the Unicode symbol must be used. For example, the degree unit matches to the Unicode degree sign (°).

  - If the unit name doesn't match to a Unicode symbol, the unit symbol must be built by Unicode Basic Latin block of characters, superscript and subscript characters. For example, watt per square meter steradian is built by `W/(m² sr)`.

  If those rules cannot be applied to the unit symbol, custom rules are allowed.

  A set of predefined unit symbols are available in `SIUnits` interface.
- Description – available as a value in `getMetadata(String)` result map. It can be requested with key `DESCRIPTION`. The property value type is `java.lang.String` and specifies a user readable description. It doesn't depend on the given unit.
- Vendor custom properties – available as a value in `getMetadata(String)` result map and can depend on the given unit. Organizations that want to use custom keys that do not clash with OSGi Working Group defined should prefix their keys in own namespace.

141.4.6 Function Property Events

The eventable function properties can trigger a new event on each property value modification. It doesn't require a modification of the value. For example, the motion sensor can send a few events with no property value change when motion is detected and continued to be detected. The event must use `FunctionEvent` class. The event properties are:

- `FUNCTION_UID` – the event source function unique identifier.
- `PROPERTY_NAME` – the property name.
- `PROPERTY_VALUE` – the property value.

For example, there is function with an eventable boolean property called “state”. When “state” value is changed to `false`, function implementation can post:

```
FunctionEvent {
    dal.function.UID=acme.function
    dal.function.property.name=”state”
    dal.function.property.value=ACMEFuntionData(java.lang.Boolean.FALSE...)
}
```

141.5 Security

141.5.1 Device Permission

The `DevicePermission` controls the bundle's authority to perform specific privileged administrative operations on the devices. There is only one action for this permission `REMOVE` to protect `remove()` method.

The name of the permission is a filter based. For more details about filter based permissions, see OSGi Core Specification, Filter Based Permissions. The filter provides an access to all device service properties. Filter attribute names are processed in a case sensitive manner. For example, the operator can give a bundle the permission to only manage devices of vendor "acme":

```
org.osgi.service.dal.DevicePermission("dal.device.hardware.vendor=acme", "remove")
```

The permission action allows the operator to assign only the necessary permissions to the bundle. For example, the management bundle can have permission to remove all registered devices:

```
org.osgi.service.dal.DevicePermission("*", "remove")
```

The code that needs to check the device permission must always use the constructor that takes the device as a parameter `Device` with a single action. For example, the implementation of `remove()` method must check that the caller has an access to the operation:

```
public class DeviceImpl implements Device {
  ...
  public void remove() {
    securityManager.checkPermission(
      new DevicePermission(this, DevicePermission.REMOVE));
  }
  ...
}
```

141.5.2 Required Permissions

The Device implementation must check the caller for the appropriate DevicePermission before execution of the remove operation. Once the DevicePermission is checked against the caller the implementation will proceed with the actual operation. The operation can require a number of other permissions to complete. The implementation must isolate the caller from such permission checks by use of proper privileged blocks.

DevicePermission check will keep the Device implementation in the call stack. This requires the implementation to have this permission to perform the operation. The security policy should be aware of this and should grant the correct permissions. Note that the DevicePermission is a filter based permission, see OSGi Core Specification, Filter Based Permissions. It provides flexibility and fine control based on the Device service properties.

141.6 org.osgi.service.dal

Device Abstraction Layer Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.dal; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.dal; version="[1.0,1.1)"

141.6.1 Summary

- Device - Represents the device in the OSGi service registry.
- DeviceException - DeviceException is a special IOException, which is thrown to indicate that there is a device operation fail.
- DevicePermission - A bundle's authority to perform specific privileged administrative operations on the devices.
- Function - Function service provides specific device operations and properties.
- FunctionData - Abstract Function data wrapper.
- FunctionEvent - Asynchronous event, which marks a function property value modification.
- OperationMetadata - Contains metadata about function operation.
- PropertyMetadata - Contains metadata about a function property, a function operation parameter or a function operation return value.
- SIUnits - Contains most of the International System of Units unit symbols.

141.6.2 public interface Device

Represents the device in the OSGi service registry. Note that Device services are registered last. Before their registration, there is Function services registration. The reverse order is used when the services are unregistered. Device services are unregistered first before Function services.

141.6.2.1 public static final String DEVICE_CATEGORY = "DAL"

Constant for the value of the Constants.DEVICE_CATEGORY service property. That category is used by all device services.

See Also Constants.DEVICE_CATEGORY

141.6.2.2 public static final String SERVICE_DESCRIPTION = "dal.device.description"

The service property value contains the device description. It's an optional property. The value type is java.lang.String.

141.6.2.3 public static final String SERVICE_DRIVER = "dal.device.driver"

The service property value contains the device driver name. For example, EnOcean, Z-Wave, Bluetooth, etc. It's a mandatory property. The value type is java.lang.String.

141.6.2.4 public static final String SERVICE_FIRMWARE_VENDOR = "dal.device.firmware.vendor"

The service property value contains the device firmware vendor. It's an optional property. The value type is java.lang.String.

141.6.2.5 public static final String SERVICE_FIRMWARE_VERSION = "dal.device.firmware.version"

The service property value contains the device firmware version. It's an optional property. The value type is java.lang.String.

141.6.2.6 public static final String SERVICE_HARDWARE_VENDOR = "dal.device.hardware.vendor"

The service property value contains the device hardware vendor. It's an optional property. The value type is java.lang.String.

141.6.2.7 public static final String SERVICE_HARDWARE_VERSION = "dal.device.hardware.version"

The service property value contains the device hardware version. It's an optional property. The value type is java.lang.String.

141.6.2.8 public static final String SERVICE_MODEL = "dal.device.model"

The service property value contains the device model. It's an optional property. The value type is java.lang.String.

141.6.2.9 public static final String SERVICE_NAME = "dal.device.name"

The service property value contains the device name. It's an optional property. The value type is java.lang.String.

141.6.2.10 public static final String SERVICE_REFERENCE_UIDS = "dal.device.reference.UIDs"

The service property value contains the reference device unique identifiers. It's an optional property. The value type is java.lang.String[]. It can be used to represent different relationships between the devices. For example, the EnOcean controller can have a reference to the USB dongle.

141.6.2.11 public static final String SERVICE_SERIAL_NUMBER = "dal.device.serial.number"

The service property value contains the device serial number. It's an optional property. The value type is java.lang.String.

141.6.2.12 public static final String SERVICE_STATUS = "dal.device.status"

The service property value contains the device status. It's a mandatory property. The value type is `java.lang.Integer`. The possible values are:

- STATUS_ONLINE
- STATUS_OFFLINE
- STATUS_REMOVED
- STATUS_PROCESSING
- STATUS_NOT_INITIALIZED
- STATUS_NOT_CONFIGURED

141.6.2.13 public static final String SERVICE_STATUS_DETAIL = "dal.device.status.detail"

The service property value contains the device status detail. It holds the reason for the current device status. It's an optional property. The value type is `java.lang.Integer`. There are two value categories:

- positive values - those values contain details related to the current status. Examples: STATUS_DETAIL_CONNECTING and STATUS_DETAIL_INITIALIZING.
- negative values - those values contain errors related to the current status. Examples: STATUS_DETAIL_CONFIGURATION_UNAPPLIED, STATUS_DETAIL_BROKEN and STATUS_DETAIL_COMMUNICATION_ERROR.

141.6.2.14 public static final String SERVICE_TYPES = "dal.device.types"

The service property value contains the device types like DVD, TV, etc. It's an optional property. The value type is `java.lang.String[]`.

141.6.2.15 public static final String SERVICE_UID = "dal.device.UID"

The service property value contains the device unique identifier. It's a mandatory property. The value type is `java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

UID ::= driver-name ':' device-id

UID - device unique identifier

driver-name - the value of the SERVICE_DRIVER service property

device-id - device unique identifier in the scope of the driver

141.6.2.16 public static final Integer STATUS_DETAIL_BROKEN

Device status detail indicates that the device is broken. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

141.6.2.17 public static final Integer STATUS_DETAIL_COMMUNICATION_ERROR

Device status detail indicates that the device communication is problematic. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_ONLINE or STATUS_NOT_INITIALIZED.

141.6.2.18 public static final Integer STATUS_DETAIL_CONFIGURATION_ERROR

Device status detail indicates that the device cannot be configured. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_NOT_CONFIGURED.

141.6.2.19 public static final Integer STATUS_DETAIL_CONFIGURATION_UNAPPLIED

Device status detail indicates that the device configuration is not applied. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_NOT_CONFIGURED.

141.6.2.20 public static final Integer STATUS_DETAIL_CONNECTING

Device status detail indicates that the device is currently connecting to the network. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

141.6.2.21 public static final Integer STATUS_DETAIL_DATA_INSUFFICIENT

Device status detail indicates that the device doesn't provide enough information and cannot be determined. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_NOT_INITIALIZED.

141.6.2.22 public static final Integer STATUS_DETAIL_DUTY_CYCLE

Device status detail indicates that the device is in duty cycle. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

141.6.2.23 public static final Integer STATUS_DETAIL_FIRMWARE_UPDATING

Device status detail indicates that the device firmware is updating. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

141.6.2.24 public static final Integer STATUS_DETAIL_INACCESSIBLE

Device status detail indicates that the device is not accessible and further communication is not possible. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_OFFLINE.

141.6.2.25 public static final Integer STATUS_DETAIL_INITIALIZING

Device status detail indicates that the device is currently in process of initialization. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

141.6.2.26 public static final Integer STATUS_DETAIL_REMOVING

Device status detail indicates that the device is leaving the network. It can be used as a value of SERVICE_STATUS_DETAIL service property. The device status must be STATUS_PROCESSING.

141.6.2.27 public static final Integer STATUS_NOT_CONFIGURED

Device status indicates that the device is currently not configured. The device can require additional actions to become completely connected to the network. It can be used as a value of SERVICE_STATUS service property.

141.6.2.28 public static final Integer STATUS_NOT_INITIALIZED

Device status indicates that the device is currently not initialized. Some protocols don't provide device information right after the device is connected. The device can be initialized later when it's awakened. It can be used as a value of SERVICE_STATUS service property.

141.6.2.29 public static final Integer STATUS_OFFLINE

Device status indicates that the device is currently not available for operations. It can be used as a value of SERVICE_STATUS service property.

141.6.2.30 public static final Integer STATUS_ONLINE

Device status indicates that the device is currently available for operations. The recent communication with the device has been passed through. It can be used as a value of SERVICE_STATUS service property.

141.6.2.31 public static final Integer STATUS_PROCESSING

Device status indicates that the device is currently busy with an operation. It can be used as a value of SERVICE_STATUS service property.

141.6.2.32 public static final Integer STATUS_REMOVED

Device status indicates that the device has been removed from the network. That status must be set as the last device status. After that the device service can be unregistered from the service registry. It can be used as a value of SERVICE_STATUS service property.

141.6.2.33 public Object getServiceProperty(String propKey)

propKey The property key.

□ Returns the current value of the specified property. The method will return the same value as ServiceReference.getProperty(String) for the service reference of this device.

This method must continue to return property values after the device service has been unregistered.

Returns The property value or null if the property key cannot be mapped to a value.

141.6.2.34 public String[] getServicePropertyKeys()

□ Returns an array with all device service property keys. The method will return the same value as ServiceReference.getPropertyKeys() for the service reference of this device. The result cannot be null.

Returns An array with all device service property keys, cannot be null.

141.6.2.35 public void remove() throws DeviceException

□ Removes this device.

The method must synchronously:

- Remove the device from the device network.
- Set the device status to STATUS_REMOVED.
- Unregister the device service from the OSGi service registry.

The caller should release the device service after successful execution, because the device will not be operational.

Throws DeviceException– If an operation error is available.

UnsupportedOperationException– If the operation is not supported over this device.

SecurityException– If the caller does not have the appropriate DevicePermission(this device, DevicePermission.REMOVE) and the Java Runtime Environment supports permissions.

IllegalStateException– If this device service object has already been unregistered.

141.6.3 public class DeviceException extends IOException

DeviceException is a special IOException, which is thrown to indicate that there is a device operation fail. The error reason can be located with getCode() method. The cause is available with getCause().

141.6.3.1 public static final int COMMUNICATION_ERROR = 1

An exception code indicates that there is an error in the communication.

141.6.3.2 public static final int NO_DATA = 4

An exception code indicates that the requested value is currently not available.

141.6.3.3 public static final int NOT_INITIALIZED = 3

An exception code indicates that the device is not initialized. The device status is Device.STATUS_NOT_INITIALIZED or Device.STATUS_PROCESSING.

141.6.3.4 public static final int TIMEOUT = 2

An exception code indicates that there is expired timeout without any processing.

141.6.3.5 public static final int UNKNOWN = 0

An exception code indicates that the error is unknown.

141.6.3.6 public DeviceException()

□ Construct a new device exception with `null` message. The cause is not initialized and the exception code is set to UNKNOWN.

141.6.3.7 public DeviceException(String message)

message The exception message.

□ Constructs a new device exception with the given message. The cause is not initialized and the exception code is set to UNKNOWN.

141.6.3.8 public DeviceException(String message, Throwable cause)

message The exception message.

cause The exception cause.

□ Constructs a new device exception with the given message and cause. The exception code is set to UNKNOWN.

141.6.3.9 public DeviceException(String message, Throwable cause, int code)

message The exception message.

cause The exception cause.

code The exception code.

□ Constructs a new device exception with the given message, cause and code.

141.6.3.10 public int getCode()

□ Returns the exception code. It indicates the reason for this exception. The code can be:

- UNKNOWN
- COMMUNICATION_ERROR
- TIMEOUT
- NOT_INITIALIZED
- NO_DATA
- custom code

Zero and positive values are reserved for this definition and further extensions of the device exception codes. Custom codes can be used only as negative values to prevent potential collisions.

Returns An exception code.

141.6.4 public class DevicePermission extends BasicPermission

A bundle's authority to perform specific privileged administrative operations on the devices. The method Device.remove() is protected with REMOVE permission action.

The name of the permission is a filter based. See OSGi Core Specification, Filter Based Permissions. The filter gives an access to all device service properties. Filter attribute names are processed in a case sensitive manner.

141.6.4.1 public static final String REMOVE = "remove"

A permission action to remove the device.

141.6.4.2 public DevicePermission(String filter, String action)

filter A filter expression that can use any device service property. The filter attribute names are processed in a case insensitive manner. A special value of "*" can be used to match all devices.

action REMOVE action.

□ Creates a new DevicePermission with the given filter and actions. The constructor must only be used to create a permission that is going to be checked.

A filter example: (dal.device.hardware.vendor=acme)

An action: remove

Throws IllegalArgumentException– If the filter syntax is not correct or invalid action is specified.

NullPointerException– If the filter or action is null.

141.6.4.3 public DevicePermission(Device device, String action)

device The device that needs to be checked for a permission.

action REMOVE action.

□ Creates a new DevicePermission with the given device and actions. The permission must be used for the security checks like:

securityManager.checkPermission(new DevicePermission(this, "remove")) . The permissions constructed by this constructor must not be added to the DevicePermission permission collections.

Throws IllegalArgumentException– If an invalid action is specified.

NullPointerException– If the device or action is null.

141.6.4.4 public boolean equals(Object obj)

obj The object being compared for equality with this object.

□ Two DevicePermission instances are equal if:

- Represents the same filter and action.
- Represents the same device (in respect to device unique identifier) and action.

Returns true if two permissions are equal, false otherwise.

141.6.4.5 public String getActions()

□ Returns the canonical string representation of REMOVE action.

Returns The canonical string representation of the actions.

141.6.4.6 public int hashCode()

□ Returns the hash code value for this object.

Returns Hash code value for this object.

141.6.4.7 public boolean implies(Permission p)

p The permission to be implied. It must be constructed by DevicePermission(Device, String).

□ Determines if the specified permission is implied by this object. The method will return false if the specified permission was not constructed by DevicePermission(Device, String). Returns true if the specified permission is a DevicePermission and this permission filter matches the specified permission device properties.

Returns true if the specified permission is implied by this permission, false otherwise.

Throws IllegalArgumentException– If the specified permission is not constructed by DevicePermission(Device, String).

141.6.4.8 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection suitable for storing DevicePermission instances.

Returns A new PermissionCollection instance.

141.6.5 public interface Function

Function service provides specific device operations and properties. Each function service must implement this interface. In additional to this interface, the implementation can provide own:

- properties;
- operations.

The function service is registered in the service registry with these service properties:

- SERVICE_UID - mandatory service property. The property value contains the function unique identifier.
- SERVICE_DEVICE_UID - optional service property. The property value is the Functional Device identifiers. The function belongs to those devices.
- SERVICE_REFERENCE_UIDS - optional service property. The property value contains the reference function unique identifiers.
- SERVICE_TYPE - mandatory service property. The property value is the function type.
- SERVICE_VERSION - optional service property. The property value contains the function version.
- SERVICE_DESCRIPTION - optional service property. The property value is the function description.
- SERVICE_OPERATION_NAMES - optional service property. The property is missing when there are no function operations and property must be set when there are function operations. The property value is the function operation names.
- SERVICE_PROPERTY_NAMES - optional service property. The property is missing when there are no function properties and property must be set when there are function properties. The property value is the function property names.

On start up, the Function services are registered before the Device services. It's possible that SERVICE_DEVICE_UID point to missing services at the moment of the registration. The reverse order is used when the services are unregistered. Function services are unregistered last after Device services.

The Function service should be registered only under the function class hierarchy. Other classes can be used if there are no ambiguous representations. For example, an ambiguous representation can be a function registered under two independent function classes like BinarySwitch and Meter. In this example, both functions support the same property state with different meaning. getPropertyMetadata(String propertyName) method cannot determinate which property is requested. It can be BinarySwitch state or Meter state.

To simplify the generic function discovery, the Function interface must be used for the service registration. In this way, the generic applications can easily find all services, which are functions in the service registry. Because of this rule, this registration is not allowed:

```
context.registerService(MeterV1.class.getName(), this, regProps);
```

If the implementation would like to mark that there is a function, but no specific function interface exists, the registration can be:

```
context.registerService(Function.class.getName(), this, regProps);
```

Note that such functions usually don't have operations and properties.

The function properties must be integrated according to these rules:

- Getter methods must be available for all properties with PropertyMetadata.ACCESS_READABLE access.
- Getter method must return a subclass of FunctionData.
- Setter methods must be available for all properties with PropertyMetadata.ACCESS_WRITABLE access.
- Setter methods can be any combination of:
  - Setter method which accepts a subclass of FunctionData.
  - Setter method which accepts the values used by the FunctionData subclass, if there are no equal types.

  It's possible to have only one or both of them. Examples:
  - There is MyFunctionData bean with BigDecimal value for a data property. Valid setters are setData(MyFunctionData data) and setData(BigDecimal data).
  - There is MySecondFunctionData bean with BigDecimal prefix and BigDecimal suffix for a data property. The prefix and suffix are using equal types and we cannot have a setter with the values used by MySecondFunctionData. The only one possible setter is setData(MySecondFunctionData data).
- No methods are required for properties with PropertyMetadata.ACCESS_EVENTABLE access.

The accessor method names must be defined according JavaBeans specification.

The function operations are java methods, which cannot override the property accessor methods. They can have zero or more parameters and zero or one return value.

Operation arguments and function properties are restricted by the same set of rules. The data type can be one of the following types:

- Java primitive type or corresponding reference type.
- java.lang.String.
- Numerical type i.e. the type which extends java.lang.Number. The numerical type must follow these conventions:
  - The type must provide a public static method called valueOf that returns an instance of the given type and takes a single String argument or a public constructor which takes a single String argument.
  - The String argument from the previous bullet can be provided by toString() method of the instance.

- Beans, but the beans properties must use those rules. Java Beans are defined in JavaBeans specification.
- java.util.Maps. The keys can be java.lang.String. The values of a single type follow these rules.
- Arrays of defined types.

The properties metadata is accessible with getPropertyMetadata(String). The operations metadata is accessible with getOperationMetadata(String).

In order to provide common behavior, all functions must follow a set of common rules related to the implementation of their setters, getters, operations and events:

- The setter method must be executed synchronously. If the underlying protocol can return response to the setter call, it must be awaited. It simplifies the property value modifications and doesn't require asynchronous callback.
- The operation method must be executed synchronously. If the underlying protocol can return an operation confirmation or response, they must be awaited. It simplifies the operation execution and doesn't require asynchronous callback.
- The getter must return the last know cached property value. The device implementation is responsible to keep that value up to date. It'll speed up the applications when the function property values are collected. The same cached value can be shared between a few requests instead of a few calls to the real device.
- The function operations, getters and setters must not override java.lang.Object and this interface methods.

141.6.5.1 public static final String SERVICE_DESCRIPTION = "dal.function.description"

The service property value contains the function description. It's an optional property. The value type is java.lang.String.

141.6.5.2 public static final String SERVICE_DEVICE_UID = "dal.function.device.UID"

The service property value contains the device unique identifier. The function belongs to this device. It's an optional property. The value type is java.lang.String.

141.6.5.3 public static final String SERVICE_OPERATION_NAMES = "dal.function.operation.names"

The service property value contains the function operation names. It's an optional property. The property is missing when there are no function operations and property must be set when there are function operations. The value type is java.lang.String[]. It's not possible to exist two or more function operations with the same name i.e. the operation overloading is not allowed.

141.6.5.4 public static final String SERVICE_PROPERTY_NAMES = "dal.function.property.names"

The service property value contains the function property names. It's an optional property. The property is missing when there are no function properties and property must be set when there are function properties. The value type is java.lang.String[]. It's not possible to exist two or more function properties with the same name.

141.6.5.5 public static final String SERVICE_REFERENCE_UIDS = "dal.function.reference.UIDs"

The service property value contains the reference function unique identifiers. It's an optional property. The value type is java.lang.String[]. It can be used to represent different relationships between the functions.

141.6.5.6 public static final String SERVICE_TYPE = "dal.function.type"

The service property value contains the function type. It's an optional property. For example, the sensor function can have different types like temperature, pressure, etc. The value type is java.lang.String.

Organizations that want to use function types that do not clash with OSGi Working Group defined types should prefix their types in own namespace.

The type doesn't mandate specific function interface. It can be used with different functions.

141.6.5.7 public static final String SERVICE_UID = "dal.function.UID"

The service property value contains the function unique identifier. It's a mandatory property. The value type is `java.lang.String`. To simplify the unique identifier generation, the property value must follow the rule:

function UID ::= device-id ':' function-id

function UID - function unique identifier

device-id - the value of the Device.SERVICE_UID Device service property

function-id - function identifier in the scope of the device

If the function is not bound to a device, the function unique identifier can be device independent.

141.6.5.8 public static final String SERVICE_VERSION = "dal.function.version"

The service property value contains the function version. That version can point to specific implementation version and vary in the different vendor implementations. It's an optional property. The value type is `java.lang.String`.

141.6.5.9 public OperationMetadata getOperationMetadata(String operationName)

operationName The function operation name, for which metadata is requested.

□ Provides metadata about the function operation.

This method must continue to return the operation metadata after the function service has been unregistered.

Returns The operation metadata for the given operation name. `null` if the operation metadata is not available.

Throws `IllegalArgumentException`– If the function operation with the specified name is not available.

141.6.5.10 public PropertyMetadata getPropertyMetadata(String propertyName)

propertyName The function property name, for which metadata is requested.

□ Provides metadata about the function property.

This method must continue to return the property metadata after the function service has been unregistered.

Returns The property metadata for the given property name. `null` if the property metadata is not available.

Throws `IllegalArgumentException`– If the function property with the specified name is not available.

141.6.5.11 public Object getServiceProperty(String propKey)

propKey The property key.

□ Returns the current value of the specified property. The method will return the same value as `ServiceReference.getProperty(String)` for the service reference of this function.

This method must continue to return property values after the device function service has been unregistered.

Returns The property value or `null` if the property key cannot be mapped to a value.

141.6.5.12 public String[] getServicePropertyKeys()

□ Returns an array with all function service property keys. The method will return the same value as ServiceReference.getPropertyKeys() for the service reference of this function. The result cannot be null.

Returns An array with all function service property keys, cannot be null.

141.6.6 public abstract class FunctionData implements Comparable<Object>

Abstract Function data wrapper. A subclass must be used for an access to the property values by all functions. It takes care about the timestamp and additional metadata. The subclasses are responsible to provide concrete value and unit if required.

141.6.6.1 public static final String DESCRIPTION = "description"

Metadata key, which value represents the data description. The property value type is java.lang.String.

141.6.6.2 public static final String FIELD_METADATA = "metadata"

Represents the metadata field name. The field value is available with getMetadata(). The field type is Map. The constant can be used as a key to FunctionData(Map).

141.6.6.3 public static final String FIELD_TIMESTAMP = "timestamp"

Represents the timestamp field name. The field value is available with getTimestamp(). The field type is long. The constant can be used as a key to FunctionData(Map).

141.6.6.4 public FunctionData(Map<String, ?> fields)

fields Contains the new FunctionData instance field values.

□ Constructs new FunctionData instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"timestamp"=Long(1384440775495)}. That map will initialize the FIELD_TIMESTAMP field with 1384440775495. If timestamp is missing, Long.MIN_VALUE is used.

- FIELD_TIMESTAMP - optional field. The value type must be Long.
- FIELD_METADATA - optional field. The value type must be Map.

Throws ClassCastException– If the field value types are not expected.

NullPointerException– If the fields map is null.

141.6.6.5 public FunctionData(long timestamp, Map<String, ?> metadata)

timestamp The data timestamp optional field.

metadata The data metadata optional field.

□ Constructs new FunctionData instance with the specified arguments.

141.6.6.6 public int compareTo(Object o)

o FunctionData to be compared.

□ Compares this FunctionData instance with the given argument. If the argument is not FunctionData, it throws ClassCastException. Otherwise, this method returns:

- -1 if this instance timestamp is less than the argument timestamp. If they are equivalent, it can be the result of the metadata map deep comparison.

- 0 if all fields are equivalent.
- 1 if this instance timestamp is greater than the argument timestamp. If they are equivalent, it can be the result of the metadata map deep comparison.

Metadata map deep comparison compares the elements of all nested java.util.Map and array instances. null is less than any other non-null instance.

Returns -1, 0 or 1 depending on the comparison rules.

Throws ClassCastException– If the method argument is not of type FunctionData or metadata maps contain values of different types for the same key.

NullPointerException– If the method argument is null.

See Also java.lang.Comparable.compareTo(java.lang.Object)

141.6.6.7 public boolean equals(Object other)

other The other instance to compare. It must be of FunctionData type.

□ Two FunctionData instances are equal if their metadata and timestamp are equivalent.

Returns true if this instance and argument have equivalent metadata and timestamp, false otherwise.

See Also java.lang.Object.equals(java.lang.Object)

141.6.6.8 public Map<String, ?> getMetadata()

□ Returns FunctionData metadata. It's dynamic metadata related only to this specific value. Possible keys:

- DESCRIPTION
- custom key

Returns FunctionData metadata or null is there is no metadata.

141.6.6.9 public long getTimestamp()

□ Returns FunctionData timestamp. The timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. The device driver is responsible to generate that value when the value is received from the device. java.lang.Long.MIN_VALUE value means no timestamp.

Returns FunctionData timestamp.

141.6.6.10 public int hashCode()

□ Returns the hash code of this FunctionData.

Returns FunctionData hash code.

See Also java.lang.Object.hashCode()

141.6.7 public class FunctionEvent extends Event

Asynchronous event, which marks a function property value modification. The event can be triggered when there is a new property value, but it's possible to have events in series with no value change. The event properties must contain:

- FUNCTION_UID - the event source function unique identifier.
- PROPERTY_NAME - the property name.

- PROPERTY_VALUE - the property value. The property value type must be a subclass of FunctionData.

141.6.7.1 public static final String EVENT_CLASS = "org/osgi/service/dal/FunctionEvent/"

Represents the event class. That constant can be useful for the event handlers depending on the event filters.

141.6.7.2 public static final String EVENT_PACKAGE = "org/osgi/service/dal/"

Represents the event package. That constant can be useful for the event handlers depending on the event filters.

141.6.7.3 public static final String FUNCTION_UID = "dal.function.UID"

Represents an event property key for function UID. The property value type is `java.lang.String`. The value represents the property value change source function identifier.

141.6.7.4 public static final String PROPERTY_NAME = "dal.function.property.name"

Represents an event property key for the function property name. The property value type is `java.lang.String`. The value represents the property name.

141.6.7.5 public static final String PROPERTY_VALUE = "dal.function.property.value"

Represents an event property key for the function property value. The property value type is a subclass of `FunctionData`. The value represents the property value.

141.6.7.6 public static final String TOPIC_PROPERTY_CHANGED = "org/osgi/service/dal/FunctionEvent/PROPERTY_CHANGED"

Represents the event topic for the function property changed.

141.6.7.7 public FunctionEvent(String topic, Dictionary<String, ?> properties)

topic The event topic.

properties The event properties.

□ Constructs a new event with the specified topic and properties.

141.6.7.8 public FunctionEvent(String topic, Map<String, ?> properties)

topic The event topic.

properties The event properties.

□ Constructs a new event with the specified topic and properties.

141.6.7.9 public FunctionEvent(String topic, String functionUID, String propName, FunctionData propValue)

topic The event topic.

functionUID The event source function UID.

propName The event source property name.

propValue The event source property value.

□ Constructs a new event with the specified topic, function UID, property name and property value.

141.6.7.10 public String getFunctionPropertyName()

□ Returns the property name. The value is same as the value of PROPERTY_NAME.

Returns The property name.

141.6.7.11 public FunctionData getFunctionPropertyValue()

□ Returns the property value. The value is same as the value of PROPERTY_VALUE.

Returns The property value.

141.6.7.12 public String getFunctionUID()

□ Returns the property value change source function identifier. The value is same as the value of FUNCTION_UID property.

Returns The property value change source function.

141.6.8 public interface OperationMetadata

Contains metadata about function operation.

See Also Function, PropertyMetadata

141.6.8.1 public static final String DESCRIPTION = "description"

Metadata key, which value represents the operation description. The property value type is `java.lang.String`.

141.6.8.2 public Map<String, ?> getMetadata()

□ Returns metadata about the function operation. The keys of the `java.util.Map` result must be of `java.lang.String` type. Possible keys:

- DESCRIPTION
- custom key

Returns The operation metadata or `null` if no such metadata is available.

141.6.8.3 public PropertyMetadata[] getParametersMetadata()

□ Returns metadata about the operation parameters or `null` if no such metadata is available.

Returns Operation parameters metadata.

141.6.8.4 public PropertyMetadata getReturnValueMetadata()

□ Returns metadata about the operation return value or `null` if no such metadata is available.

Returns Operation return value metadata.

141.6.9 public interface PropertyMetadata

Contains metadata about a function property, a function operation parameter or a function operation return value. The access to the function properties is a bitmap value of ACCESS metadata key. Function properties can be accessed in three ways. Any combinations between them are possible:

- ACCESS_READABLE - available for all properties, which can be read. Function must provide a getter method for an access to the property value.
- ACCESS_WRITABLE - available for all properties, which can be modified. Function must provide a setter method for a modification of the property value.
- ACCESS_EVENTABLE - available for all properties, which can report the property value. FunctionEvents are sent on property change.

See Also Function, PropertyMetadata

141.6.9.1 public static final String ACCESS = "access"

Metadata key, which value represents the access to the function property. The property value is a bitmap of `Integer` type. The bitmap can be any combination of:

- ACCESS_READABLE
- ACCESS_WRITABLE
- ACCESS_EVENTABLE

For example, value Integer(3) means that the property is readable and writable, but not eventable.

The property access is available only for function properties and it's missing for the operation parameters.

141.6.9.2 public static final int ACCESS_EVENTABLE = 4

Marks the eventable function properties. The flag can be used as a part of bitmap value of ACCESS.

See Also Function

141.6.9.3 public static final int ACCESS_READABLE = 1

Marks the readable function properties. The flag can be used as a part of bitmap value of ACCESS. The readable access mandates function to provide a property getter method.

See Also Function

141.6.9.4 public static final int ACCESS_WRITABLE = 2

Marks the writable function properties. The flag can be used as a part of bitmap value of ACCESS. The writable access mandates function to provide a property setter methods.

See Also Function

141.6.9.5 public static final String DESCRIPTION = "description"

Metadata key, which value represents the property description. The property value type is java.lang.String.

141.6.9.6 public static final String UNITS = "units"

Metadata key, which value represents the property supported units. The property value type is java.lang.String[]. The array first element at index 0 represents the default unit. Each unit must follow those rules:

- The International System of Units must be used where it's applicable. For example, kg for kilogram and km for kilometer.
- If the unit name matches to an Unicode symbol name, the Unicode symbol must be used. For example, the degree unit matches to the Unicode degree sign (°).
- If the unit name doesn't match to an Unicode symbol, the unit symbol must be built by Unicode Basic Latin block of characters, superscript and subscript characters. For example, watt per square meter steradian is built by W/(m² sr).

If those rules cannot be applied to the unit symbol, custom rules are allowed. A set of predefined unit symbols are available in SIUnits interface.

141.6.9.7 public FunctionData[] getEnumValues(String unit)

unit The unit to align the supported values, can be null.

□ Returns the property possible values according to the specified unit. If the unit is null, the values set is aligned to the default unit. If there is no such set of supported values, null is returned. The values must be sorted in increasing order.

Returns The supported values according to the specified unit or null if no such values are supported. The values must be sorted in increasing order.

Throws IllegalArgumentException– If the unit is not supported.

141.6.9.8 public FunctionData getMaxValue(String unit)

unit The unit to align the maximum value, can be null .

□ Returns the property maximum value according to the specified unit. If the unit is null, the maximum value is aligned to the default unit. If there is no maximum value, null is returned.

Returns The maximum value according to the specified unit or null if no maximum value is supported.

Throws IllegalArgumentException– If the unit is not supported.

141.6.9.9 public Map<String, ?> getMetadata(String unit)

unit The unit to align the metadata if it's applicable. It can be null, which means that the default unit will be used.

□ Returns metadata about the function property or operation parameter. The keys of the java.util.Map result must be of java.lang.String type. Possible keys:

- DESCRIPTION - doesn't depend on the given unit.
- ACCESS - available only for function property and missing for function operation parameters. It doesn't depend on the given unit.
- UNITS - doesn't depend on the given unit.
- custom key - can depend on the unit. Organizations that want to use custom keys that do not clash with OSGi Working Group defined should prefix their keys in own namespace.

Returns The property metadata or null if no such metadata is available.

141.6.9.10 public FunctionData getMinValue(String unit)

unit The unit to align the minimum value, can be null .

□ Returns the property minimum value according to the specified unit. If the unit is null, the minimum value is aligned to the default unit. If there is no minimum value, null is returned.

Returns The minimum value according to the specified unit or null if no minimum value is supported.

Throws IllegalArgumentException– If the unit is not supported.

141.6.9.11 public FunctionData getStep(String unit)

unit The unit to align the step, can be null.

□ Returns the difference between two values in series. For example, if the range is [0, 100], the step can be 10.

Returns The step according to the specified unit or null if no step is supported.

Throws IllegalArgumentException– If the unit is not supported.

141.6.10 public final class SIUnits

Contains most of the International System of Units unit symbols. The constant name represents the unit name. The constant value represents the unit symbol as it's defined in PropertyMetadata.UNITS.

141.6.10.1 public static final String AMPERE = "A"

Unit of electric current defined by the International System of Units (SI). It's one of be base units called ampere.

141.6.10.2 public static final String AMPERE_PER_METER = "A/m"

Unit of magnetic field strength. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called ampere per meter.

141.6.10.3 **public static final String AMPERE_PER_SQUARE_METER = "A/m\u00b2"**

Unit of current density. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called ampere per square meter.

141.6.10.4 **public static final String ANGSTROM = "\u212b"**

Unit of length. It's one of other non-SI units. The unit is called angstrom.

141.6.10.5 **public static final String BAR = "bar"**

Unit of pressure. It's one of other non-SI units. The unit is called bar.

141.6.10.6 **public static final String BARN = "b"**

Unit of area. It's one of other non-SI units. The unit is called barn.

141.6.10.7 **public static final String BECQUEREL = "Bq"**

Unit of activity referred to a radionuclide. It's one of the coherent derived units in the SI with special names and symbols. The unit is called becquerel.

141.6.10.8 **public static final String BEL = "B"**

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called bel.

141.6.10.9 **public static final String CANDELA = "cd"**

Unit of luminous intensity defined by the International System of Units (SI). It's one of be base units called candela.

141.6.10.10 **public static final String CANDELA_PER_SQUARE_METER = "cd/m\u00b2"**

Unit of luminance. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called candela per square meter.

141.6.10.11 **public static final String COULOMB = "C"**

Unit of electronic charge, amount of electricity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called coulomb.

141.6.10.12 **public static final String COULOMB_PER_CUBIC_METER = "C/m\u00b3"**

Unit of electric charge density. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per cubic meter.

141.6.10.13 **public static final String COULOMB_PER_KILOGRAM = "C/kg"**

Unit of exposure (x- and gamma-rays). It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per kilogram.

141.6.10.14 **public static final String COULOMB_PER_SQUARE_METER = "C/m\u00b2"**

Unit of surface charge density, electric flux density, electric displacement. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called coulomb per square meter.

141.6.10.15 **public static final String CUBIC_METER = "m\u00b3"**

Unit of volume. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called cubic meter.

141.6.10.16 **public static final String CUBIC_METER_PER_KILOGRAM = "m\u00b3/kg"**

Unit of specific volume. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called cubic meter per kilogram.

141.6.10.17 **public static final String DAY = "d"**

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called day.

141.6.10.18 **public static final String DECIBEL = "dB"**

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called decibel.

141.6.10.19 **public static final String DEGREE = "\u00b0"**

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called degree.

141.6.10.20 **public static final String DEGREE_CELSIUS = "\u2103"**

Unit of Celsius temperature. It's one of the coherent derived units in the SI with special names and symbols. The unit is called degree Celsius.

141.6.10.21 **public static final String DYNE = "dyn"**

Unit of force. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called dyne.

141.6.10.22 **public static final String ERG = "erg"**

Unit of energy. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called erg.

141.6.10.23 **public static final String FARAD = "F"**

Unit of capacitance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called farad.

141.6.10.24 **public static final String FARAD_PER_METER = "F/m"**

Unit of permittivity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called farad per meter.

141.6.10.25 **public static final String GAL = "Gal"**

Unit of acceleration. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called gal.

141.6.10.26 **public static final String GAUSS = "G"**

Unit of magnetic flux density. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called gauss.

141.6.10.27 **public static final String GRAY = "Gy"**

Unit of absorbed dose, specific energy (imparted), kerma. It's one of the coherent derived units in the SI with special names and symbols. The unit is called gray.

141.6.10.28 **public static final String GRAY_PER_SECOND = "Gy/s"**

Unit of absorbed dose rate. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called gray per second.

141.6.10.29 public static final String HECTARE = "ha"

Unit of area. It's one of non-SI units accepted for use with the International System of Units. The unit is called hectare.

141.6.10.30 public static final String HENRY = "H"

Unit of inductance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called henry.

141.6.10.31 public static final String HENRY_PER_METER = "H/m"

Unit of permeability. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called henry per meter.

141.6.10.32 public static final String HERTZ = "Hz"

Unit of frequency. It's one of the coherent derived units in the SI with special names and symbols. The unit is called hertz.

141.6.10.33 public static final String HOUR = "h"

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called hour.

141.6.10.34 public static final String JOULE = "J"

Unit of energy, work, amount of electricity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called joule.

141.6.10.35 public static final String JOULE_PER_CUBIC_METER = "J/m\u00b3"

Unit of energy density. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per cubic meter.

141.6.10.36 public static final String JOULE_PER_KELVIN = "J/\u212a"

Unit of heat capacity, entropy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kelvin.

141.6.10.37 public static final String JOULE_PER_KILOGRAM = "J/kg"

Unit of specific energy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kilogram.

141.6.10.38 public static final String JOULE_PER_KILOGRAM_KELVIN = "J/(kg \u212a)"

Unit of specific heat capacity, specific entropy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per kilogram kelvin.

141.6.10.39 public static final String JOULE_PER_MOLE = "J/mol"

Unit of molar energy. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per mole.

141.6.10.40 public static final String JOULE_PER_MOLE_KELVIN = "J/(mol \u212a)"

Unit of molar entropy, molar heat capacity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called joule per mole kelvin.

141.6.10.41 public static final String KATAL = "kat"

Unit of catalytic activity. It's one of the coherent derived units in the SI with special names and symbols. The unit is called katal.

141.6.10.42 **public static final String KATAL_PER_CUBIC_METER = "kat/m\u00b3"**

Unit of catalytic activity concentration. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called katal per cubic meter.

141.6.10.43 **public static final String KELVIN = "\u212a"**

Unit of thermodynamic temperature defined by the International System of Units (SI). It's one of be base units called kelvin.

141.6.10.44 **public static final String KILOGRAM = "kg"**

Unit of mass defined by the International System of Units (SI). It's one of be base units called kilogram.

141.6.10.45 **public static final String KILOGRAM_PER_CUBIC_METER = "kg/m\u00b3"**

Unit of density, mass density, mass concentration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called kilogram per cubic meter.

141.6.10.46 **public static final String KILOGRAM_PER_SQUARE_METER = "kg/m\u00b2"**

Unit of surface density. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called kilogram per square meter.

141.6.10.47 **public static final String KNOT = "kn"**

Unit of speed. It's one of other non-SI units. The unit is called knot.

141.6.10.48 **public static final String LITER = "l"**

Unit of volume. It's one of non-SI units accepted for use with the International System of Units. The unit is called liter. International System of Units accepts two symbols: lower-case l and capital L. That constant value is using the lower-case l.

141.6.10.49 **public static final String LUMEN = "lm"**

Unit of luminous flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called lumen.

141.6.10.50 **public static final String LUX = "lx"**

Unit of illuminance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called lux.

141.6.10.51 **public static final String MAXWELL = "Mx"**

Unit of magnetic flux. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called maxwell.

141.6.10.52 **public static final String METER = "m"**

Unit of length defined by the International System of Units (SI). It's one of be base units called meter.

141.6.10.53 **public static final String METER_PER_SECOND = "m/s"**

Unit of speed, velocity. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called meter per second.

141.6.10.54 **public static final String METER_PER_SECOND_SQUARED = "m/s\u00b2"**

Unit of acceleration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called meter per second squared.

141.6.10.55 public static final String MILLIMETER_OF_MERCURY = "mmHg"

Unit of pressure. It's one of other non-SI units. The unit is called millimeter of mercury.

141.6.10.56 public static final String MOLE = "mol"

Unit of amount of substance defined by the International System of Units (SI). It's one of be base units called mole.

141.6.10.57 public static final String MOLE_PER_CUBIC_METER = "mol/m\u00b3"

Unit of amount concentration, concentration. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called mole per cubic meter.

141.6.10.58 public static final String NAUTICAL_MILE = "M"

Unit of distance. It's one of other non-SI units. The unit is called nautical mile.

141.6.10.59 public static final String NEPER = "Np"

Unit of logarithmic ratio quantities. It's one of other non-SI units. The unit is called neper.

141.6.10.60 public static final String NEWTON = "N"

Unit of force. It's one of the coherent derived units in the SI with special names and symbols. The unit is called newton.

141.6.10.61 public static final String NEWTON_METER = "N m"

Unit of moment of force. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called newton meter.

141.6.10.62 public static final String NEWTON_PER_METER = "N/m"

Unit of surface tension. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called newton per meter.

141.6.10.63 public static final String OERSTED = "Oe"

Unit of magnetic field. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called oersted.

141.6.10.64 public static final String OHM = "\u2126"

Unit of electric resistance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called ohm.

141.6.10.65 public static final String PASCAL = "Pa"

Unit of pressure, stress. It's one of the coherent derived units in the SI with special names and symbols. The unit is called pascal.

141.6.10.66 public static final String PASCAL_SECOND = "Pa s"

Unit of dynamic viscosity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called pascal second.

141.6.10.67 public static final String PHOT = "ph"

Unit of illuminance. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called phot.

141.6.10.68 public static final String PLANE_ANGLE_MINUTE = "\u2032"

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called minute.

141.6.10.69 **public static final String PLANE_ANGLE_SECOND = "\u2033"**

Unit of plane angle. It's one of non-SI units accepted for use with the International System of Units. The unit is called second.

141.6.10.70 **public static final String POISE = "P"**

Unit of dynamic viscosity. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called poise.

141.6.10.71 **public static final String PREFIX_ATTO = "a"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called atto and represents the 18th negative power of ten.

141.6.10.72 **public static final String PREFIX_CENTI = "c"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called centi and represents the 2nd negative power of ten.

141.6.10.73 **public static final String PREFIX_DECA = "da"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called deca and represents the 1st power of ten.

141.6.10.74 **public static final String PREFIX_DECI = "d"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called deci and represents the 1st negative power of ten.

141.6.10.75 **public static final String PREFIX_EXA = "E"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called exa and represents the 18th power of ten.

141.6.10.76 **public static final String PREFIX_FEMTO = "f"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called femto and represents the 15th negative power of ten.

141.6.10.77 **public static final String PREFIX_GIGA = "G"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called giga and represents the 9th power of ten.

141.6.10.78 **public static final String PREFIX_HECTO = "h"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called hecto and represents the 2nd power of ten.

141.6.10.79 **public static final String PREFIX_KILO = "k"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called kilo and represents the 3rd power of ten.

141.6.10.80 **public static final String PREFIX_MEGA = "M"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called mega and represents the 6th power of ten.

141.6.10.81 **public static final String PREFIX_MICRO = "\u00b5"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called micro and represents the 6th negative power of ten.

141.6.10.82 **public static final String PREFIX_MILLI = "m"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called milli and represents the 3rd negative power of ten.

141.6.10.83 **public static final String PREFIX_NANO = "n"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called nano and represents the 9th negative power of ten.

141.6.10.84 **public static final String PREFIX_PICO = "p"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called pico and represents the 12th negative power of ten.

141.6.10.85 **public static final String PREFIX_YOCTO = "y"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called yocto and represents the 24th negative power of ten.

141.6.10.86 **public static final String PREFIX_YOTTA = "Y"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called yotta and represents the 24th power of ten.

141.6.10.87 **public static final String PREFIX_ZEPTO = "z"**

Adopted prefix symbol to form the symbols of the decimal submultiples of SI units. It's called zepto and represents the 21th negative power of ten.

141.6.10.88 **public static final String PREFIX_ZETTA = "Z"**

Adopted prefix symbol to form the symbols of the decimal multiples of SI units. It's called zetta and represents the 21th power of ten.

141.6.10.89 **public static final String RADIAN = "rad"**

Unit of plane angle. It's one of the coherent derived units in the SI with special names and symbols. The unit is called radian.

141.6.10.90 **public static final String RADIAN_PER_SECOND = "rad/s"**

Unit of angular velocity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called radian per second.

141.6.10.91 **public static final String RADIAN_PER_SECOND_SQUARED = "rad/s\u00b2"**

Unit of angular acceleration. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called radian per second squared.

141.6.10.92 **public static final String RECIPROCAL_METER = "m\u207b\u00b9"**

Unit of wavenumber. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called reciprocal meter.

141.6.10.93 **public static final String SECOND = "s"**

Unit of time defined by the International System of Units (SI). It's one of be base units called second.

141.6.10.94 **public static final String SIEMENS = "S"**

Unit of electric conductance. It's one of the coherent derived units in the SI with special names and symbols. The unit is called siemens.

141.6.10.95 **public static final String SIEVERT = "Sv"**

Unit of dose equivalent, ambient dose equivalent, directional dose equivalent, personal dose equivalent. It's one of the coherent derived units in the SI with special names and symbols. The unit is called sievert.

141.6.10.96 **public static final String SQUARE_METER = "m\u00b2"**

Unit of area. It's one of coherent derived units in the SI expressed in terms of base units. The unit is called square meter.

141.6.10.97 **public static final String STERADIAN = "sr"**

Unit of solid angle. It's one of the coherent derived units in the SI with special names and symbols. The unit is called steradian.

141.6.10.98 **public static final String STILB = "sb"**

Unit of luminance. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called stilb.

141.6.10.99 **public static final String STOKES = "St"**

Unit of kinematic viscosity. It's one of non-SI units associated with the CGS and the CGS-Gaussian system of units. The unit is called stokes.

141.6.10.100 **public static final String TESLA = "T"**

Unit of magnetic flux density. It's one of the coherent derived units in the SI with special names and symbols. The unit is called tesla.

141.6.10.101 **public static final String TIME_MINUTE = "min"**

Unit of time. It's one of non-SI units accepted for use with the International System of Units. The unit is called minute.

141.6.10.102 **public static final String TONNE = "t"**

Unit of mass. It's one of non-SI units accepted for use with the International System of Units. The unit is called tonne.

141.6.10.103 **public static final String VOLT = "V"**

Unit of electric potential difference, electromotive force. It's one of the coherent derived units in the SI with special names and symbols. The unit is called volt.

141.6.10.104 **public static final String VOLT_PER_METER = "V/m"**

Unit of electric field strength. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called volt per meter.

141.6.10.105 **public static final String WATT = "W"**

Unit of power, radiant flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called watt.

141.6.10.106 **public static final String WATT_PER_METER_KELVIN = "W/(m \u212a)"**

Unit of thermal conductivity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per meter kelvin.

141.6.10.107 **public static final String WATT_PER_SQUARE_METER = "W/m\u00b2"**

Unit of heat flux density, irradiance. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per square meter.

141.6.10.108 public static final String WATT_PER_SQUARE_METER_STERADIAN = "W/(m\u00b2 sr)"

Unit of radiance. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per square meter steradian.

141.6.10.109 public static final String WATT_PER_STERADIAN = "W/sr"

Unit of radiant intensity. It's one of coherent derived units whose names and symbols include SI coherent derived units with special names and symbols. The unit is called watt per steradian.

141.6.10.110 public static final String WEBER = "Wb"

Unit of magnetic flux. It's one of the coherent derived units in the SI with special names and symbols. The unit is called weber.

141.7 References

[1] *JavaBeans Spec*
https://www.oracle.com/java/technologies/javase/javabeans-spec.html

142 Device Abstraction Layer Functions Specification

Version 1.0

142.1 Introduction

Concrete function interfaces are used to unify the access and the control of the basic device operations and the related properties. The current section specifies the minimal set of such functionalities. They can be extended or replaced to cover domain specific scenarios. The set is not closed and can be incorporated with vendor specific functions. There is support for: control, monitoring and metering information.

142.2 Functions

142.2.1 BooleanControl

BooleanControl function provides a binary control support. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856. The full function definition is available in the next tables.

Table 142.1 BooleanControl Operations

Name	Description
inverse	Reverses the BooleanControl state. If the current state represents true value, it'll be changed to false. If the current state represents false value, it'll be changed to true.
setTrue	Sets the BooleanControl state to true value.
setFalse	Sets the BooleanControl state to false value.

Table 142.2 BooleanControl Properties

Name	Description
data	Contains the current state of BooleanControl. The property access is readable, writable and eventable.

Different types can be used as a value of SERVICE_TYPE service property. The next list contains some suitable to BooleanControl:

- LIGHT - indicates that there is a light device control. true state means that the light device will be turned on. false state means that the light device will be turned off.
- DOOR - indicates that there is a door position control. true state means that the door will be opened. false state means that the door will be closed.

- WINDOW - indicates that there is a window position control. true state means that the window will be opened. false state means that the window will be closed.
- POWER - indicates that there is electricity control. true state means that the power will be restored. false state means that the power will be cut.
- other type defined in Types
- vendor specific

The function is using *BooleanData* on page 888 data structure to provide the control state.

The next code snippet sets to true all BooleanControl functions, which control the light.

```
ServiceReference[] booleanControlSRefs = context.getServiceReferences(
      BooleanControl.class.getName(),
      '(' + Function.SERVICE_TYPE + '=' + Types.LIGHT + ')');
if (booleanControlSRefs != null) {
  for (int i = 0; i < booleanControlSRefs.length; i++) {
    BooleanControl booleanControl = (BooleanControl) context.getService(
        booleanControlSRefs[i]);
    if (booleanControl != null) {
      booleanControl.setTrue();
      context.ungetService(booleanControlSRefs[i]);
    }
  }
}
```

142.2.2 BooleanSensor

BooleanSensor function provides binary sensor monitoring. It reports the state when an important event is available. There are no operations. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856. The full function definition is available in the next table.

Table 142.3 *BooleanSensor Properties*

Name	Description
data	Contains the current state of BooleanSensor. The property access is readable and eventable.

Different types can be used as a value of SERVICE_TYPE service property. The next list contains some suitable to BooleanSensor:

- LIGHT - indicates that the BooleanSensor can detected light. true state means that there is light. false state means that there is no light.
- GAS - indicates that the BooleanSensor supports gas detection. true state means there is gas. false state means that there is no gas.
- SMOKE - indicates that the BooleanSensor can detect smoke. true state means that there is smoke. false state means that there is no smoke.
- DOOR - indicates that the BooleanSensor can detect the door state. true state means that the door is opened. false state means that the door is closed.
- WINDOW - indicates that the BooleanSensor can window state. true state means that the window is opened. false state means that the window is closed.
- POWER - indicates that the BooleanSensor can detect power/no power. true state means that there is power. false state means that there is no power.
- RAIN - indicates that the BooleanSensor can detect rain. true state means that there is rain. false state means that there is no rain.

- CONTACT - indicates that the BooleanSensor can detect contact. true state means that there is contact. false state means that there is no contact.
- FIRE - indicates that the BooleanSensor can detect fire. true state means that there is fire. false state means that there is no fire.
- OCCUPANCY - indicates that the BooleanSensor can detect presence. true state means that someone is detected. false state means that nobody is detected.
- WATER - indicates that the BooleanSensor can detect water leak. true state means that there is water leak. false state means that there is no water leak.
- MOTION - indicates that the BooleanSensor can detect motion. true state means that there is motion detection. false state means that there is no motion detection.
- other type defined in Types
- vendor specific

The function is using *BooleanData* on page 888 data structure to provide the sensor state.

142.2.3 MultiLevelControl

MultiLevelControl function provides multi-level control support. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856. The full function definition is available in the next table.

Table 142.4 *MultiLevelControl Properties*

Name	Description
data	Contains the current state of MultiLevelControl. The property access is readable, writable and eventable.

Different types can be used as a value of SERVICE_TYPE service property. The next list contains some suitable to MultiLevelControl:

- LIGHT - indicates that the MultiLevelControl can control light devices. Usually, such devices are called dimmable. MultiLevelControl minimum value can switch off the device and MultiLevelControl maximum value can increase the device light to the maximum possible value.
- TEMPERATURE - indicates that the MultiLevelControl can control temperature devices. For example, such device can be thermostat. MultiLevelControl minimum value is the lowest supported temperature. MultiLevelControl maximum value is the highest supported temperature.
- FLOW - indicates that the MultiLevelControl can control the flow level. MultiLevelControl minimum value is the minimum supported flow level. MultiLevelControl maximum value is the maximum supported flow level.
- PRESSURE - indicates that the MultiLevelControl can control the pressure level. MultiLevelControl minimum value is the lowest supported pressure level. MultiLevelControl maximum value is the highest supported pressure level.
- HUMIDITY - indicates that the MultiLevelControl can control the humidity level. It's typical functionality for HVAC (heating, ventilation, and air conditioning) devices. MultiLevelControl minimum value is the lowest supported humidity level. MultiLevelControl maximum value is the highest supported humidity level.
- GAS - indicates that the MultiLevelControl can control the gas level. MultiLevelControl minimum value is the lowest supported gas level. MultiLevelControl maximum value is the highest supported gas level.
- SMOKE - indicates that the MultiLevelControl can control the smoke level. MultiLevelControl minimum value is the lowest supported smoke level. MultiLevelControl maximum value is the highest supported smoke level.

- DOOR - indicates that the MultiLevelControl can control the door position. MultiLevelControl minimum value can completely close the door. MultiLevelControl maximum value can open the door to the maximum allowed position.
- WINDOW - indicates that the MultiLevelControl can control the window position. MultiLevel-Control minimum value can completely close the window. MultiLevelControl maximum value can open the window to the maximum allowed position.
- LIQUID - indicates that the MultiLevelControl can control the liquid level. MultiLevelControl minimum value is the lowest supported liquid level. MultiLevelControl maximum value is the highest supported liquid level.
- POWER - indicates that the MultiLevelControl can control the power level. MultiLevelControl minimum value is the lowest supported power level. MultiLevelControl maximum value is the highest supported power level.
- NOISINESS - indicates that the MultiLevelControl can control the noise level. MultiLevelControl minimum value is the lowest supported noise level. MultiLevelControl maximum value is the highest supported noise level.
- other type defined in Types
- vendor specific

The function is using *LevelData* on page 889 data structure to provide the level.

142.2.4 MultiLevelSensor

MultiLevelSensor function provides multi-level sensor monitoring. It reports its state when an important event is available. There are no operations. The property eventing must follow the definition of Device Abstraction Later, *Function Property Events* on page 856. The full function definition is available in the next table.

Table 142.5 *MultiLevelSensor Properties*

Name	Description
data	Contains the current state of MultiLevelSensor. The property access is readable and eventable.

Different types can be used as a value of SERVICE_TYPE service property. The next list contains some suitable to MultiLevelSensor:

- LIGHT - indicates that the sensor can monitor the light level.
- TEMPERATURE - indicates that the sensor can monitor the temperature.
- FLOW - indicates that the sensor can monitor the flow level.
- PRESSURE - indicates that the sensor can monitor the pressure level.
- HUMIDITY - indicates that the sensor can monitor the humidity level.
- GAS - indicates that the sensor can monitor the gas level.
- SMOKE - indicates that the sensor can monitor the smoke level.
- DOOR - indicates that the sensor can monitor the door position.
- WINDOW - indicates that the sensor can monitor the window position.
- LIQUID - indicates that the sensor can monitor the liquid level.
- POWER - indicates that the sensor can monitor the power level.
- NOISINESS - indicates that the sensor can monitor the noise level.
- RAIN - indicates that the MultiLevelSensor can monitor the rain rate.
- other type defined in Types
- vendor specific

The function is using *LevelData* on page 889 data structure to provide the level.

142.2.5 Meter

Meter function can measure metering information. It provides the current and total consumptions or generations. The property eventing must follow the definition of Device Abstraction Later, *Function Property Events* on page 856. The full function definition is available in the next tables.

Table 142.6 Meter Properties

Name	Description
total	Contains the total consumption or production. The property access is readable and eventable.
current	Contains the current consumption or production. The property access is readable and eventable.

Different types can be used as a value of SERVICE_TYPE service property. The next list contains some suitable to Meter:

- PRESSURE - indicates that the Meter measures pressure.
- GAS - indicates that the Meter measures the gas consumption.
- POWER - indicates that the Meter measures the power consumption.
- WATER - indicates that the Meter measures water consumption.
- HEAT - indicates that the Meter measures thermal energy provided by a source.
- COLD - indicates that the Meter measures thermal energy provided by a source.
- other type defined in Types
- vendor specific

The function is using *LevelData* on page 889 data structure to provide metering information.

Meter function service can be optionally registered with SERVICE_FLOW service property. The value type is java.lang.String. It contains the metering flow. Currently, the flow can be FLOW_IN for a consumption or FLOW_OUT for a production.

142.2.6 Alarm

Alarm function provides alarm sensor support. There is only one eventable property and no operations. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856. The full function definition is available in the next table.

Table 142.7 BooleanSensor Properties

Name	Description
alarm	Specifies the alarm property name. The property is eventable.

The function is using *AlarmData* on page 889 data structure to report the alarm. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856.

142.2.7 Keypad

Keypad function provides support for keypad control. The keypad typically consists of one or more keys/buttons, which can be discerned. Different types of key presses like short and long press can typically also be detected. Each key pressed event is followed by a key released event. It's not possible to have two consecutive key pressed or key released events. There is only one eventable property and no operations. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856. The full function definition is available in the next table.

Table 142.8 *Keypad Properties*

Name	Description
key	Specifies a property name for a key from the keypad. The property is eventable.

The function is using *KeypadData* on page 889 data structure to report the keys.

142.2.8 WakeUp

WakeUp function provides device awake monitoring. It's especially applicable to battery-operated devices. Such device can notify the system that it's awake and can receive commands with a PROPERTY_AWAKE property event. The property eventing must follow the definition of Device Abstraction Layer, *Function Property Events* on page 856.

The device can periodically wake up for commands. The interval can be managed with PROPERTY_WAKE_UP_INTERVAL property.

Table 142.9 *WakeUp Properties*

Name	Description
awake	Specifies the awake eventable property name. If the device is awake, it will trigger a property event. The property value type is *BooleanData* on page 888.
wakeUpInterval	Specifies the wake up interval. The device can periodically wake up and receive commands. That interval is managed by this property. The property access is readable, writable and eventable. The property value type is *LevelData* on page 889.

142.3 Functions Data

FunctionData subclasses are wrappers on top of the java types to cover the requirements of the Device Abstraction Layer section. They can be received with the getter methods, can be set with the setter methods and can be reported with FunctionEvent. The value can be described with different properties like:

- timestamp - the timestamp is the difference between the value collecting time and midnight, January 1, 1970 UTC. It's measured in milliseconds. The device driver is responsible to generate that value when the value is received from the device.
- unit - represents the value unit as it's defined in *Function Properties* on page 854.
- description - represents a human readable description of the value.

142.3.1 BooleanData

BooleanData is used by *BooleanControl* on page 883, *BooleanSensor* on page 884 and *WakeUp* on page 888.

It provides information about the function state. That data object contains boolean value, the value collecting time and additional metadata. The value field is accessible with getValue() getter. Other fields are inherited from the parent class FunctionData.

Two BooleanData instances are equal if they contain equal metadata, timestamp and boolean value.

compareTo(Object) method compares BooleanData instance with the given argument of the same type and returns:

- -1 if the instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if the instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, value.

142.3.2 LevelData

LevelData is used by *MultiLevelControl* on page 885, *MultiLevelSensor* on page 886, *Meter* on page 887 and *WakeUp* on page 888.

It provides information about the function level. That data object contains BigDecimal value and the value unit. The measurement unit is used as it's defined in *Function Properties* on page 854. The unit field is accessible with getUnit() getter. The level field is accessible with getLevel() getter.

Two LevelData instances are equal if they contain equal metadata, timestamp, unit and level.

compareTo(Object) method compares LevelData instance with the given argument of the same type and returns:

- -1 if the instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if the instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, level, unit.

142.3.3 AlarmData

AlarmData is used by *Alarm* on page 887.

AlarmData data structure is used to provide information about the available alarm. That data object contains:

- alarm type - indicates the meaning of the alarm like smoke, power fail, etc.
- alarm severity - indicates the alarm importance level like minor, critical, etc.

The severity field is accessible with getSeverity() getter. The type field is accessible with getType() getter.

Two AlarmData instances are equal if they contain equal metadata, timestamp, type and severity.

compareTo(Object) method compares AlarmData instance with the given argument of the same type and returns:

- -1 if the instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if the instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, type, severity.

142.3.4 KeypadData

KeypadData is used by *Keypad* on page 887.

KeypadData data structure is used to provide information when a change with some key from the keypad has occurred. That data object contains the event type, sub-type, key code and key name. Currently, there are two predefined event types:

- TYPE_PRESSED – used for a key pressed;
- TYPE_RELEASED – used for a key released.

Predefined event sub-types are:

- SUB_TYPE_PRESSED_NORMAL – used for a normal key pressed event. Usually, there is a single press and the key is not held down. This sub-type is used with TYPE_PRESSED type.
- SUB_TYPE_PRESSED_LONG – used for a long key pressed event. Usually, there is a single press and the key is held down. This sub-type is used with TYPE_PRESSED type.
- SUB_TYPE_PRESSED_DOUBLE – used for a double key pressed event. Usually, there are two press actions and the key is not held down after the second press. This sub-type is used with TYPE_PRESSED type.
- SUB_TYPE_PRESSED_DOUBLE_LONG – used for a double long key pressed event. Usually, there are two press actions and the key is held down after the second press. This sub-type is used with TYPE_PRESSED type.

The type field is accessible with getType() getter. The subType field is accessible with getSubType() getter. The keyCode field is accessible with getKeyCode() getter. The keyName field is accessible with getKeyName() getter.

Two KeypadData instances are equal if they contain equal metadata, timestamp, event type, sub-type, key code and key name.

compareTo(Object) method compares KeypadData instance with the given argument of the same type and returns:

- -1 if the instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if the instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, type, sub-type, key code, key name.

142.4 org.osgi.service.dal.functions

Device Abstraction Layer Functions Package 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dal.functions; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dal.functions; version="[1.0,1.1)"
```

142.4.1 Summary

- Alarm - Alarm function provides alarm sensor support.
- BooleanControl - BooleanControl function provides a boolean control support.
- BooleanSensor - BooleanSensor function provides boolean sensor monitoring.
- Keypad - Keypad function provides support for keypad control.
- Meter - Meter function can measure metering information.
- MultiLevelControl - MultiLevelControl function provides multi-level control support.
- MultiLevelSensor - MultiLevelSensor function provides multi-level sensor monitoring.
- Types - Shares common constants for all functions defined in this package.
- WakeUp - WakeUp function provides device awake monitoring.

142.4.2 public interface Alarm extends Function

Alarm function provides alarm sensor support. There is only one eventable property and no operations.

See Also AlarmData

142.4.2.1 public static final String PROPERTY_ALARM = "alarm"

Specifies the alarm property name. The property is eventable.

See Also AlarmData

142.4.3 public interface BooleanControl extends Function

BooleanControl function provides a boolean control support. The eventable function state is accessible with getData() getter and setData(boolean) setter. The state can be reversed with inverse() method, can be set to true value with setTrue() method and can be set to false value with setFalse() method.

The control type can be:

- Types.LIGHT
- Types.DOOR
- Types.WINDOW
- Types.POWER
- other type defined in Types
- custom - vendor specific type

See Also BooleanData

142.4.3.1 public static final String OPERATION_INVERSE = "inverse"

Specifies the inverse operation name. The operation can be executed with inverse() method.

142.4.3.2 public static final String OPERATION_SET_FALSE = "setFalse"

Specifies the operation name, which sets the control state to false value. The operation can be executed with setFalse() method.

142.4.3.3 public static final String OPERATION_SET_TRUE = "setTrue"

Specifies the operation name, which sets the control state to true value. The operation can be executed with setTrue() method.

142.4.3.4 public static final String PROPERTY_DATA = "data"

Specifies the state property name. The eventable property value is accessible with getData() method.

See Also BooleanData

142.4.3.5 public BooleanData getData() throws DeviceException

□ Returns the current state of BooleanControl. It's a getter method for PROPERTY_DATA property.

Returns The current state of BooleanControl.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

See Also BooleanData, BooleanControl.PROPERTY_DATA

142.4.3.6 public void inverse() throws DeviceException

□ Reverses the BooleanControl state. If the current state represents true value, it'll be changed to false. If the current state represents false value, it'll be changed to true. The operation name is OPERATION_INVERSE.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

142.4.3.7 public void setData(boolean data) throws DeviceException

data The new function value.

□ Sets the BooleanControl state to the specified value. It's setter method for PROPERTY_DATA property.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

IllegalArgumentException– If there is an invalid argument.

See Also BooleanControl.PROPERTY_DATA

142.4.3.8 public void setFalse() throws DeviceException

□ Sets the BooleanControl state to false value. The operation name is OPERATION_SET_FALSE.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

142.4.3.9 public void setTrue() throws DeviceException

□ Sets the BooleanControl state to true value. The operation name is OPERATION_SET_TRUE.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

142.4.4 public interface BooleanSensor extends Function

BooleanSensor function provides boolean sensor monitoring. It reports its state when an important event is available. The eventable state is accessible with getData() getter. There are no operations.

The sensor type can be:

- Types.LIGHT
- Types.GAS
- Types.SMOKE
- Types.DOOR
- Types.WINDOW
- Types.POWER
- Types.RAIN
- Types.CONTACT
- Types.FIRE
- Types.OCCUPANCY
- Types.WATER
- Types.MOTION
- other type defined in Types
- custom - vendor specific type

See Also BooleanData

142.4.4.1 public static final String PROPERTY_DATA = "data"

Specifies the state property name. The eventable property value is accessible with getData() getter.

142.4.4.2 public BooleanData getData() throws DeviceException

□ Returns the BooleanSensor current state. It's a getter method for PROPERTY_DATA property.

Returns The BooleanSensor current state.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

See Also BooleanData

142.4.5 public interface Keypad extends Function

Keypad function provides support for keypad control. The keypad typically consists of one or more keys/buttons, which can be discerned. Different types of key presses like short and long press can typically also be detected. Each key pressed event is followed by a key released event. It's not possible to have two consecutive key pressed or key released events. There is only one eventable property and no operations.

Keypad can enumerate all supported keys in the key property metadata, PropertyMetadata.getEnumValues(String).

See Also KeypadData

142.4.5.1 public static final String PROPERTY_KEY = "key"

Specifies a property name for a key from the keypad. The property is eventable.

See Also KeypadData

142.4.6 public interface Meter extends Function

Meter function can measure metering information. The function provides these properties:

- PROPERTY_CURRENT - eventable property accessible with getCurrent() getter;
- PROPERTY_TOTAL - eventable property accessible with getTotal() getter.

The sensor type can be:

- Types.PRESSURE
- Types.GAS
- Types.POWER
- Types.WATER
- Types.HEAT
- Types.COLD
- other type defined in Types
- custom - vendor specific type

See Also LevelData

142.4.6.1 public static final String FLOW_IN = "in"

Represents the metering consumption flow. It can be used as SERVICE_FLOW property value.

142.4.6.2 public static final String FLOW_OUT = "out"

Represents the metering production flow. It can be used as SERVICE_FLOW property value.

142.4.6.3 public static final String PROPERTY_CURRENT = "current"

Specifies the current consumption or production property name. The eventable property can be read with getCurrent() getter.

142.4.6.4 public static final String PROPERTY_TOTAL = "total"

Specifies the total consumption or production property name. The eventable property can be read with getTotal() getter.

142.4.6.5 public static final String SERVICE_FLOW = "dal.meter.flow"

The service property value contains the metering flow. It's an optional property and available only if it's supported by the meter. The value type is `java.lang.String`. Possible property values:

- FLOW_IN
- FLOW_OUT

142.4.6.6 public LevelData getCurrent() throws DeviceException

□ Returns the current metering info. It's a getter method for PROPERTY_CURRENT property.

Returns The current metering info.

Throws `IllegalStateException`– If this function service object has already been unregistered.

`DeviceException`– If an operation error is available.

See Also LevelData

142.4.6.7 public LevelData getTotal() throws DeviceException

□ Returns the total metering info. It's a getter method for PROPERTY_TOTAL property.

Returns The total metering info.

Throws `IllegalStateException`– If this function service object has already been unregistered.

`DeviceException`– If an operation error is available.

See Also LevelData

142.4.7 public interface MultiLevelControl extends Function

`MultiLevelControl` function provides multi-level control support. The eventable function level is accessible with getData() getter and setData(BigDecimal, String) setter.

The control type can be:

- Types.LIGHT
- Types.TEMPERATURE
- Types.FLOW
- Types.PRESSURE
- Types.HUMIDITY
- Types.GAS
- Types.SMOKE
- Types.DOOR
- Types.WINDOW

- Types.LIQUID
- Types.POWER
- Types.NOISINESS
- other type defined in Types
- custom - vendor specific type

See Also LevelData

142.4.7.1 public static final String PROPERTY_DATA = "data"

Specifies the level property name. The eventable property can be read with getData() getter and can be set with setData(BigDecimal, String) setters.

142.4.7.2 public LevelData getData() throws DeviceException

□ Returns MultiLevelControl level. It's a getter method for PROPERTY_DATA property.

Returns MultiLevelControl level.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

See Also LevelData

142.4.7.3 public void setData(BigDecimal level, String unit) throws DeviceException

level The new control level.

unit The level unit.

□ Sets MultiLevelControl level according to the specified unit. It's a setter method for PROPERTY_DATA property.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

IllegalArgumentException– If there is an invalid argument.

142.4.8 public interface MultiLevelSensor extends Function

MultiLevelSensor function provides multi-level sensor monitoring. It reports its state when an important event is available. The eventable state is accessible with getData() getter. There are no operations.

The sensor type can be:

- Types.LIGHT
- Types.TEMPERATURE
- Types.FLOW
- Types.PRESSURE
- Types.HUMIDITY
- Types.GAS
- Types.SMOKE
- Types.DOOR
- Types.WINDOW
- Types.LIQUID
- Types.POWER
- Types.NOISINESS

- Types.RAIN
- other type defined in Types
- custom - vendor specific type

See Also LevelData

142.4.8.1 public static final String PROPERTY_DATA = "data"

Specifies the state property name. The eventable property can be read with getData() getter.

See Also LevelData

142.4.8.2 public LevelData getData() throws DeviceException

□ Returns the MultiLevelSensor current state. It's a getter method for PROPERTY_DATA property.

Returns The MultiLevelSensor current state.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

See Also LevelData

142.4.9 public interface Types

Shares common constants for all functions defined in this package. The defined function types are mapped as follow:

- LIGHT - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- TEMPERATURE - MultiLevelControl and MultiLevelSensor
- FLOW - MultiLevelControl and MultiLevelSensor
- PRESSURE - MultiLevelControl, MultiLevelSensor and Meter
- HUMIDITY - MultiLevelControl and MultiLevelSensor
- GAS - MultiLevelControl, MultiLevelSensor, BooleanSensor and Meter
- SMOKE - MultiLevelControl, MultiLevelSensor and BooleanSensor
- DOOR - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- WINDOW - MultiLevelControl, MultiLevelSensor, BooleanSensor and BooleanControl
- LIQUID - MultiLevelControl and MultiLevelSensor
- POWER - MultiLevelControl, MultiLevelSensor, BooleanSensor, BooleanControl and Meter
- NOISINESS - MultiLevelControl and MultiLevelSensor
- RAIN - MultiLevelSensor and BooleanSensor
- CONTACT - BooleanSensor
- FIRE - BooleanSensor
- OCCUPANCY - BooleanSensor
- WATER - BooleanSensor and Meter
- MOTION - BooleanSensor
- HEAT - Meter
- COLD - Meter

The mapping is not mandatory. The function can use custom defined types.

142.4.9.1 public static final String COLD = "cold"

The function type is applicable to:

- Meter - indicates that the Meter measures thermal energy provided by a source.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.2 public static final String CONTACT = "contact"

The function type is applicable to:

- BooleanSensor - indicates that the BooleanSensor can detect contact. true state means that there is contact. false state means that there is no contact.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.3 public static final String DOOR = "door"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the door position. MultiLevelControl minimum value can completely close the door. MultiLevelControl maximum value can open the door to the maximum allowed position.
- MultiLevelSensor - indicates that the sensor can monitor the door position.
- BooleanSensor - indicates that the BooleanSensor can detect the door state. true state means that the door is opened. false state means that the door is closed.
- BooleanControl - indicates that there is a door position control. true state means that the door will be opened. false state means that the door will be closed.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.4 public static final String FIRE = "fire"

The function type is applicable to:

- BooleanSensor - indicates that the BooleanSensor can detect fire. true state means that there is fire. false state means that there is no fire.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.5 public static final String FLOW = "flow"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the flow level. MultiLevelControl minimum value is the minimum supported flow level. MultiLevelControl maximum value is the maximum supported flow level.
- MultiLevelSensor - indicates that the sensor can monitor the flow level.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.6 public static final String GAS = "gas"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the gas level. MultiLevelControl minimum value is the lowest supported gas level. MultiLevelControl maximum value is the highest supported gas level.
- MultiLevelSensor - indicates that the sensor can monitor the gas level.
- BooleanSensor - indicates that the BooleanSensor supports gas detection. true state means there is gas. false state means that there is no gas.
- Meter - indicates that the Meter measures the gas consumption.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.7 public static final String HEAT = "heat"

The function type is applicable to:

- Meter - indicates that the Meter measures thermal energy provided by a source.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.8 public static final String HUMIDITY = "humidity"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the humidity level. It's typical functionality for HVAC (heating, ventilation, and air conditioning) devices. MultiLevelControl minimum value is the lowest supported humidity level. MultiLevelControl maximum value is the highest supported humidity level.
- MultiLevelSensor - indicates that the sensor can monitor the humidity level.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.9 public static final String LIGHT = "light"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control light devices. Usually, such devices are called dimmable. MultiLevelControl minimum value can switch off the device and MultiLevelControl maximum value can increase the device light to the maximum possible value.
- MultiLevelSensor - indicates that the sensor can monitor the light level.
- BooleanSensor - indicates that the BooleanSensor can detected light. true state means that there is light. false state means that there is no light.
- BooleanControl - indicates that there is a light device control. true state means that the light device will be turned on. false state means that the light device will be turned off.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.10 public static final String LIQUID = "liquid"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the liquid level. MultiLevelControl minimum value is the lowest supported liquid level. MultiLevelControl maximum value is the highest supported liquid level.
- MultiLevelSensor - indicates that the sensor can monitor the liquid level.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.11 public static final String MOTION = "motion"

The function type is applicable to:

- BooleanSensor - indicates that the BooleanSensor can detect motion. true state means that there is motion detection. false state means that there is no motion detection.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.12 public static final String NOISINESS = "noisiness"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the noise level. MultiLevelControl minimum value is the lowest supported noise level. MultiLevelControl maximum value is the highest supported noise level.

- MultiLevelSensor - indicates that the sensor can monitor the noise level.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.13 public static final String OCCUPANCY = "occupancy"

The function type is applicable to:

- BooleanSensor - indicates that the BooleanSensor can detect presence. true state means that someone is detected. false state means that nobody is detected.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.14 public static final String POWER = "power"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the power level. MultiLevelControl minimum value is the lowest supported power level. MultiLevelControl maximum value is the highest supported power level.
- MultiLevelSensor - indicates that the sensor can monitor the power level.
- BooleanSensor - indicates that the BooleanSensor can detect power/no power. true state means that there is power. false state means that there is no power.
- BooleanControl - indicates that there is electricity control. true state means that the power will be restored. false state means that the power will be cut.
- Meter - indicates that the Meter measures the power consumption.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.15 public static final String PRESSURE = "pressure"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the pressure level. MultiLevelControl minimum value is the lowest supported pressure level. MultiLevelControl maximum value is the highest supported pressure level.
- MultiLevelSensor - indicates that the sensor can monitor the pressure level.
- Meter - Indicates that the Meter measures pressure.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.16 public static final String RAIN = "rain"

The function type is applicable to:

- MultiLevelSensor - indicates that the MultiLevelSensor can monitor the rain rate. It's not applicable to MultiLevelControl.
- BooleanSensor - indicates that the BooleanSensor can detect rain. true state means that there is rain. false state means that there is no rain.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.17 public static final String SMOKE = "smoke"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the smoke level. MultiLevelControl minimum value is the lowest supported smoke level. MultiLevelControl maximum value is the highest supported smoke level.

- MultiLevelSensor - indicates that the sensor can monitor the smoke level.
- BooleanSensor - indicates that the BooleanSensor can detect smoke. true state means that there is smoke. false state means that there is no smoke.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.18 public static final String TEMPERATURE = "temperature"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control temperature devices. For example, such device can be thermostat. MultiLevelControl minimum value is the lowest supported temperature. MultiLevelControl maximum value is the highest supported temperature.
- MultiLevelSensor - indicates that the sensor can monitor the temperature.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.19 public static final String WATER = "water"

The function type is applicable to:

- BooleanSensor - indicates that the BooleanSensor can detect water leak. true state means that there is water leak. false state means that there is no water leak.
- Meter - indicates that the Meter measures water consumption.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.9.20 public static final String WINDOW = "window"

The function type is applicable to:

- MultiLevelControl - indicates that the MultiLevelControl can control the window position. MultiLevelControl minimum value can completely close the window. MultiLevelControl maximum value can open the window to the maximum allowed position.
- MultiLevelSensor - indicates that the sensor can monitor the window position.
- BooleanSensor - indicates that the BooleanSensor can window state. true state means that the window is opened. false state means that the window is closed.
- BooleanControl - indicates that there is a window position control. true state means that the window will be opened. false state means that the window will be closed.

This type can be specified as a value of org.osgi.service.dal.Function.SERVICE_TYPE.

142.4.10 public interface WakeUp extends Function

WakeUp function provides device awake monitoring. It's especially applicable to battery-operated devices. Such device can notify the system that it's awake and can receive commands with a PROPERTY_AWAKE property event.

The device can periodically wake up for commands. The interval can be managed with PROPERTY_WAKE_UP_INTERVAL property.

See Also LevelData, BooleanData

142.4.10.1 public static final String PROPERTY_AWAKE = "awake"

Specifies the awake property name. The property access type can be PropertyMetadata.ACCESS_EVENTABLE. If the device is awake, it will trigger a property event.

The property value type is BooleanData. The boolean data is always true. It marks that the device is awake.

142.4.10.2 public static final String PROPERTY_WAKE_UP_INTERVAL = "wakeUpInterval"

Specifies the wake up interval. The device can periodically wake up and receive commands. That interval is managed by this eventable property. The current property value is available with getWakeUpInterval() and can be modified with setWakeUpInterval(BigDecimal, String).

142.4.10.3 public LevelData getWakeUpInterval() throws DeviceException

□ Returns the current wake up interval. It's a getter method for PROPERTY_WAKE_UP_INTERVAL property. The device can periodically wake up and receive command based on this interval.

The interval can be measured in different units like hours, minutes, seconds, etc. The unit is specified in LevelData instance.

Returns The current wake up interval.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

See Also LevelData

142.4.10.4 public void setWakeUpInterval(BigDecimal interval, String unit) throws DeviceException

interval The new wake up interval.

unit The interval unit. If the unit is null, the interval is measured in milliseconds.

□ Sets wake up interval according to the specified unit. It's a setter method for PROPERTY_WAKE_UP_INTERVAL property. The device can periodically wake up and receive command based on this interval. The unit can be null, then the interval is measured in milliseconds.

Throws IllegalStateException– If this function service object has already been unregistered.

DeviceException– If an operation error is available.

IllegalArgumentException– If there is an invalid argument.

142.5 org.osgi.service.dal.functions.data

Device Abstraction Layer Functions Data Package 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.dal.functions.data; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.dal.functions.data; version="[1.0,1.1)"

142.5.1 Summary

- AlarmData - Function alarm data.
- BooleanData - Function boolean data wrapper.
- KeypadData - Represents a keypad event data that is collected when a change with some key from the keypad has occurred.

- LevelData - Function level data wrapper.

142.5.2 public class AlarmData extends FunctionData

Function alarm data. It cares about the alarm type, severity, timestamp and additional metadata. It doesn't support unit. The alarm type is mapped to FunctionData value.

See Also Alarm, FunctionData

142.5.2.1 public static final String FIELD_SEVERITY = "severity"

Represents the severity field name. The field value is available with getSeverity(). The field type is int. The constant can be used as a key to AlarmData(Map) .

142.5.2.2 public static final String FIELD_TYPE = "type"

Represents the type field name. The field value is available with getType(). The field type is int. The constant can be used as a key to AlarmData(Map).

142.5.2.3 public static final int SEVERITY_CRITICAL = 3

The severity rating indicates that there a critical alarm. The severity priority is higher than SEVERITY_MINOR and SEVERITY_MAJOR.

142.5.2.4 public static final int SEVERITY_MAJOR = 2

The severity rating indicates that there is a major alarm. The severity priority is higher than SEVERITY_MINOR and lower than SEVERITY_CRITICAL.

142.5.2.5 public static final int SEVERITY_MINOR = 1

The severity rating indicates that there is a minor alarm. The severity priority is lower than SEVERITY_MAJOR and SEVERITY_CRITICAL.

142.5.2.6 public static final int SEVERITY_UNDEFINED = 0

The severity constant indicates that there is no severity rating for this alarm.

142.5.2.7 public static final int TYPE_ACCESS_CONTROL = 1

The alarm type indicates that there is access control issue. For example, the alarm can indicate that the door is unlocked.

142.5.2.8 public static final int TYPE_BURGLAR = 2

The alarm type indicates that there is a burglar notification. For example, the alarm can indicate that the glass is broken.

142.5.2.9 public static final int TYPE_COLD = 3

The alarm type indicates that temperature is too low.

142.5.2.10 public static final int TYPE_GAS_CO = 4

The alarm type indicates that carbon monoxide (CO) is detected.

142.5.2.11 public static final int TYPE_GAS_CO2 = 5

The alarm type indicates that carbon dioxide (CO2) is detected.

142.5.2.12 public static final int TYPE_HARDWARE_FAIL = 7

The alarm type indicates that there is hardware failure.

142.5.2.13 public static final int TYPE_HEAT = 6

The alarm type indicates that temperature is too high.

142.5.2.14 public static final int TYPE_POWER_FAIL = 8

The alarm type indicates a power cut.

142.5.2.15 public static final int TYPE_SMOKE = 9

The alarm type indicates that smoke is detected.

142.5.2.16 public static final int TYPE_SOFTWARE_FAIL = 10

The alarm type indicates that there is software failure.

142.5.2.17 public static final int TYPE_TAMPER = 11

The alarm type for a tamper indication.

142.5.2.18 public static final int TYPE_UNDEFINED = 0

The alarm type indicates that the type is not specified.

142.5.2.19 public static final int TYPE_WATER = 12

The alarm type indicates that a water leak is detected.

142.5.2.20 public AlarmData(Map<String, ?> fields)

fields Contains the new AlarmData instance field values.

□ Constructs new AlarmData instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"severity"=Integer(1)...}. That map will initialize the FIELD_SEVERITY field with 1. If severity is missing, SEVERITY_UNDEFINED is used.

- FIELD_SEVERITY - optional field. The value type must be Integer.
- FIELD_TYPE - optional field. The value type must be Integer.

Throws ClassCastException– If the field value types are not expected.

IllegalArgumentException– If the alarm severity is invalid.

NullPointerException– If the fields map is null.

142.5.2.21 public AlarmData(long timestamp, Map<String, ?> metadata, int severity, int type)

timestamp The alarm data timestamp optional field.

metadata The alarm data metadata optional field.

severity The alarm data severity optional field.

type The alarm data type optional field.

□ Constructs new AlarmData instance with the specified arguments.

Throws IllegalArgumentException– If the alarm severity is invalid.

142.5.2.22 public int compareTo(Object o)

o AlarmData to be compared.

□ Compares this AlarmData instance with the given argument. If the argument is not AlarmData, it throws ClassCastException. Otherwise, this method returns:

- -1 if this instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if this instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, type, severity.

Returns -1, 0 or 1 depending on the comparison rules.

Throws ClassCastException– If the method argument is not of type AlarmData.

See Also java.lang.Comparable.compareTo(java.lang.Object)

142.5.2.23 public boolean equals(Object o)

o The object to compare this data.

□ Two AlarmData instances are equal if they contain equal metadata, timestamp, type and severity.

Returns true if this object is equivalent to the specified one.

See Also org.osgi.service.dal.FunctionData.equals(java.lang.Object)

142.5.2.24 public int getSeverity()

□ Returns the alarm severity. The severity can be one of:

- SEVERITY_UNDEFINED
- SEVERITY_MINOR
- SEVERITY_MAJOR
- SEVERITY_CRITICAL

Returns The alarm severity.

142.5.2.25 public int getType()

□ Returns the alarm type. The type can be one of the predefined:

- TYPE_UNDEFINED
- TYPE_SMOKE
- TYPE_HEAT
- TYPE_COLD
- TYPE_GAS_CO
- TYPE_GAS_CO2
- TYPE_WATER
- TYPE_POWER_FAIL
- TYPE_HARDWARE_FAIL
- TYPE_SOFTWARE_FAIL
- vendor specific

Zero and positive values are reserved for this definition and further extensions of the alarm types. Custom types can be used only as negative values to prevent potential collisions.

Returns The alarm type.

142.5.2.26 public int hashCode()

□ Returns the hash code for this AlarmData object. The hash code is a sum of FunctionData.hashCode(), the alarm severity and the alarm type.

Returns The hash code of this AlarmData object.

See Also org.osgi.service.dal.FunctionData.hashCode()

142.5.2.27 public String toString()

□ Returns the string representation of this alarm data.

Returns The string representation of this alarm data.

142.5.3 public class BooleanData extends FunctionData

Function boolean data wrapper. It can contain a boolean value, timestamp and additional metadata. It doesn't support measurement unit.

See Also BooleanControl, BooleanSensor, FunctionData

142.5.3.1 public static final String FIELD_VALUE = "value"

Represents the value field name. The field value is available with getValue(). The field type is boolean . The constant can be used as a key to BooleanData(Map).

142.5.3.2 public BooleanData(Map<String, ?> fields)

fields Contains the new BooleanData instance field values.

□ Constructs new BooleanData instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"value"=Boolean(true)...}. That map will initialize the FIELD_VALUE field with true.

FIELD_VALUE - mandatory field. The value type must be Boolean.

Throws ClassCastException– If the field value types are not expected.

IllegalArgumentException– If the value is missing.

NullPointerException– If the fields map is null.

142.5.3.3 public BooleanData(long timestamp, Map<String, ?> metadata, boolean value)

timestamp The boolean data timestamp optional field.

metadata The boolean data metadata optional field.

value The boolean value mandatory field.

□ Constructs new BooleanData instance with the specified arguments.

142.5.3.4 public int compareTo(Object o)

o BooleanData to be compared.

□ Compares this BooleanData instance with the given argument. If the argument is not BooleanData, it throws ClassCastException. Otherwise, this method returns:

- -1 if this instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if this instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, value.

Returns -1, 0 or 1 depending on the comparison rules.

Throws ClassCastException– If the method argument is not of type BooleanData.

See Also java.lang.Comparable.compareTo(java.lang.Object)

142.5.3.5 public boolean equals(Object o)

o The object to compare this data.

□ Two BooleanData instances are equal if they contain equal metadata, timestamp and boolean value.

Returns true if this object is equivalent to the specified one.

See Also org.osgi.service.dal.FunctionData.equals(java.lang.Object)

142.5.3.6 public boolean getValue()

□ Returns BooleanData value.

Returns BooleanData value.

142.5.3.7 public int hashCode()

□ Returns the hash code for this BooleanData object. The hash code is a sum of FunctionData.hashCode() and Boolean.hashCode(), where Boolean.hashCode() represents the boolean value hash code.

Returns The hash code of this BooleanData object.

See Also org.osgi.service.dal.FunctionData.hashCode()

142.5.3.8 public String toString()

□ Returns the string representation of this boolean data.

Returns The string representation of this boolean data.

142.5.4 public class KeypadData extends FunctionData

Represents a keypad event data that is collected when a change with some key from the keypad has occurred.

The key pressed event is using TYPE_PRESSED type, while the key released event is using TYPE_RELEASED type.

See Also Keypad, FunctionData

142.5.4.1 public static final String FIELD_KEY_CODE = "keyCode"

Represents the key code field name. The field value is available with getKeyCode(). The field type is int . The constant can be used as a key to KeypadData(Map).

142.5.4.2 public static final String FIELD_KEY_NAME = "keyName"

Represents the key name field name. The field value is available with getKeyName(). The field type is String. The constant can be used as a key to KeypadData(Map).

142.5.4.3 public static final String FIELD_SUB_TYPE = "subType"

Represents the event sub-type field name. The field value is available with getSubType(). The field type is int. The constant can be used as a key to KeypadData(Map).

142.5.4.4 public static final String FIELD_TYPE = "type"

Represents the event type field name. The field value is available with getType(). The field type is int. The constant can be used as a key to KeypadData(Map).

142.5.4.5 public static final int SUB_TYPE_PRESSED_DOUBLE = 3

Represents a keypad event sub-type for a double key pressed event. Usually, there are two press actions and the key is not held down after the second press. This sub-type is used with TYPE_PRESSED type.

142.5.4.6 public static final int SUB_TYPE_PRESSED_DOUBLE_LONG = 4

Represents a keypad event sub-type for a double long key pressed event. Usually, there are two press actions and the key is held down after the second press. This sub-type is used with TYPE_PRESSED type.

142.5.4.7 public static final int SUB_TYPE_PRESSED_LONG = 2

Represents a keypad event sub-type for a long key pressed event. Usually, there is a single press and the key is held down. This sub-type is used with TYPE_PRESSED type.

142.5.4.8 public static final int SUB_TYPE_PRESSED_NORMAL = 1

Represents a keypad event sub-type for a normal key pressed event. Usually, there is a single press and the key is not held down. This sub-type is used with TYPE_PRESSED type.

142.5.4.9 public static final int TYPE_PRESSED = 0

Represents a keypad event type for a key pressed event.

142.5.4.10 public static final int TYPE_RELEASED = 1

Represents a keypad event type for a key released event.

142.5.4.11 public KeypadData(Map<String, ?> fields)

fields Contains the new KeypadData instance field values.

□ Constructs new KeypadData instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"type"=Integer(1)...}. That map will initialize the FIELD_TYPE field with 1.

- FIELD_TYPE - mandatory field. The value type must be Integer.
- FIELD_SUB_TYPE - optional field. The value type must be Integer.
- FIELD_KEY_CODE - mandatory field. The value type must be Integer.
- FIELD_KEY_NAME - optional field. The value type must be String.

Throws ClassCastException– If the field value types are not expected.

IllegalArgumentException– If the event type or key code is missing or invalid arguments are specified.

NullPointerException– If the fields map is null.

142.5.4.12 public KeypadData(long timestamp, Map<String, Object> metadata, int type, int subType, int keyCode, String keyName)

timestamp The data timestamp optional field.

metadata The data metadata optional field.

type The data event type mandatory field.

subType The data event sub-type optional field or 0 if there is no sub-type.

keyCode The data key code mandatory field.

keyName The data key name optional field or null if there is no key name.

□ Constructs new KeypadData instance with the specified arguments.

142.5.4.13 public int compareTo(Object o)

o KeypadData to be compared.

□ Compares this KeypadData instance with the given argument. If the argument is not KeypadData, it throws ClassCastException. Otherwise, this method returns:

- -1 if this instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if this instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, type, sub-type, key code, key name.

Returns -1, 0 or 1 depending on the comparison rules.

Throws ClassCastException– If the method argument is not of type KeypadData.

See Also java.lang.Comparable.compareTo(java.lang.Object)

142.5.4.14 public boolean equals(Object o)

o The object to compare this data.

□ Two KeypadData instances are equal if they contain equal metadata, timestamp, event type, key code and key name.

Returns true if this object is equivalent to the specified one.

See Also org.osgi.service.dal.FunctionData.equals(java.lang.Object)

142.5.4.15 public int getKeyCode()

□ The code of the key. This field is mandatory and it holds the semantics(meaning) of the key.

Returns The key code.

142.5.4.16 public String getKeyName()

□ Represents a human readable name of the corresponding key code. This field is optional and sometimes it could be missed(might be null).

Returns A string with the name of the key or null if not specified.

142.5.4.17 public int getSubType()

□ Returns the event sub-type. The sub-type provides additional details about the event. The sub-type can be one of:

- SUB_TYPE_PRESSED_NORMAL
- SUB_TYPE_PRESSED_LONG
- SUB_TYPE_PRESSED_DOUBLE
- SUB_TYPE_PRESSED_DOUBLE_LONG
- custom sub-type

Zero and positive values are reserved for this definition and further extensions of the sub-types. Custom sub-types can be used only as negative values to prevent potential collisions.

Returns The event sub-type.

142.5.4.18 public int getType()

□ Returns the event type. The type represents the main reason for this event. It can be one of:

- TYPE_PRESSED

- TYPE_RELEASED

Returns The event type.

142.5.4.19 public int hashCode()

□ Returns the hash code for this KeypadData object. The hash code is a sum of FunctionData.hashCode(), String.hashCode(), event type, event sub-type and key code, where String.hashCode() represents the key name hash code if available.

Returns The hash code of this LevelData object.

See Also org.osgi.service.dal.FunctionData.hashCode()

142.5.4.20 public String toString()

□ Returns the string representation of this keypad data.

Returns The string representation of this keypad data.

142.5.5 public class LevelData extends FunctionData

Function level data wrapper. It supports all properties defined in FunctionData.

See Also MultiLevelControl, MultiLevelSensor, Meter, FunctionData

142.5.5.1 public static final String FIELD_LEVEL = "level"

Represents the level field name. The field value is available with getLevel(). The field type is BigDecimal. The constant can be used as a key to LevelData(Map).

142.5.5.2 public static final String FIELD_UNIT = "unit"

Represents the unit field name. The field value is available with getUnit(). The field type is String. The constant can be used as a key to LevelData(Map).

142.5.5.3 public LevelData(Map<String, ?> fields)

fields Contains the new LevelData instance field values.

□ Constructs new LevelData instance with the specified field values. The map keys must match to the field names. The map values will be assigned to the appropriate class fields. For example, the maps can be: {"level"=BigDecimal(1)...}. That map will initialize the FIELD_LEVEL field with 1.

- FIELD_LEVEL - mandatory field. The value type must be BigDecimal.
- FIELD_UNIT - optional field. The value type must be String.

Throws ClassCastException– If the field value types are not expected.

IllegalArgumentException– If the level is missing.

NullPointerException– If the fields map is null.

142.5.5.4 public LevelData(long timestamp, Map<String, Object> metadata, BigDecimal level, String unit)

timestamp The data timestamp optional field.

metadata The data metadata optional field.

level The level value mandatory field.

unit The data unit optional field.

□ Constructs new LevelData instance with the specified arguments.

Throws NullPointerException– If level is null.

142.5.5.5 public int compareTo(Object o)

o LevelData to be compared.

□ Compares this LevelData instance with the given argument. If the argument is not LevelData, it throws ClassCastException. Otherwise, this method returns:

- -1 if this instance field is less than a field of the specified argument.
- 0 if all fields are equivalent.
- 1 if this instance field is greater than a field of the specified argument.

The fields are compared in this order: timestamp, metadata, level, unit.

Returns -1, 0 or 1 depending on the comparison rules.

Throws ClassCastException– If the method argument is not of type LevelData.

See Also java.lang.Comparable.compareTo(java.lang.Object)

142.5.5.6 public boolean equals(Object o)

o The object to compare this data.

□ Two LevelData instances are equal if they contain equal metadata, timestamp, unit and level.

Returns true if this object is equivalent to the specified one.

See Also org.osgi.service.dal.FunctionData.equals(java.lang.Object)

142.5.5.7 public BigDecimal getLevel()

□ Returns LevelData value. The value type is BigDecimal instead of double to guarantee value accuracy.

Returns The LevelData value.

142.5.5.8 public String getUnit()

□ Returns LevelData unit as it's specified in PropertyMetadata.UNITS or null if the unit is missing.

Returns The value unit or null if the unit is missing.

142.5.5.9 public int hashCode()

□ Returns the hash code for this LevelData object. The hash code is a sum of FunctionData.hashCode(), String.hashCode() and BigDecimal.hashCode(), where String.hashCode() represents the unit hash code and BigDecimal.hashCode() represents the level hash code.

Returns The hash code of this LevelData object.

See Also org.osgi.service.dal.FunctionData.hashCode()

142.5.5.10 public String toString()

□ Returns the string representation of this level data.

Returns The string representation of this level data.

143 Network Interface Information Service Specification

Version 1.0

143.1 Introduction

The Network Interface Information Service is a service that provides a standard way for bundles to receive notification about changes in the network interface and IP address.

When the IP address has changed, bundles utilizing the IP address information need to detect this change. When using the standard Java API, such as `java.net.NetworkInterface` and `java.net.InetAddress`, calls to confirm the IP address at regular intervals are required. Since this is a process common to all bundles that need to detect any change in IP address information, this specification defines a notification feature for all available network interfaces, including the IP address. In addition, this specification defines an API that provides the function to obtain the network interface information and the information about the IP address bound to a network interface.

The name of a network interface can be Operating System specific. In order for bundles to refer to the network interface it is necessary to distinguish the network interface in a form that it is independent of the Operating System.

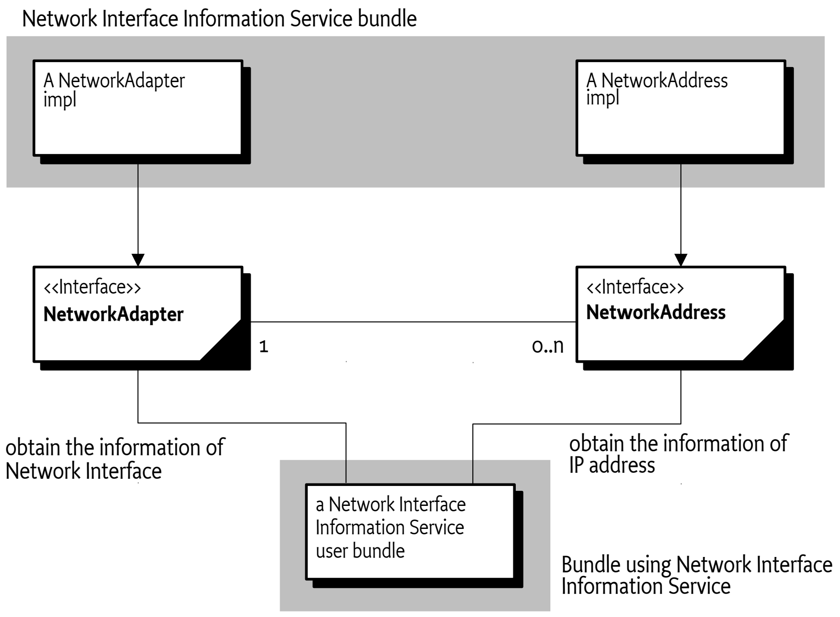
This specification defines the `NetworkAdapter` Service and `NetworkAddress` Service. These services provide information about the network interface and IP addresses.

143.1.1 Entities

- *Network Interface* - Available and activated network interfaces provided in the execution environment. In this specification, the unit of the network interface is the logical interface, not the physical interface.
- *NetworkAdapter* - The OSGi service that provides information related to the Network Interface. This service provides function similar to `java.net.NetworkInterface`.
- *NetworkAddress* - The OSGi service that provides information of IP addresses available on the execution environment in which a Network Interface Information Service bundle is running.
- *Network Interface Information Service bundle* - The OSGi bundle that implements NetworkAdapter and NetworkAddress services. Network Interface Information Service bundle registers NetworkAdapter and NetworkAddress services with the Framework.
- *Network Interface Type* - An identifier of the network interface. It is independent of the operating system. The two type of identifier string is specified in this specification. This specification allows that Network Interface type other than them can be defined by the platform provider in each environment. This identifier is used by user bundle to specify the network interface to be monitored.
- *IPAddressVersion* - An identifier indicating the IP address version. For example, IPv4, IPv6. This identifier is defined in this specification. This identifier is used by a bundle to specify the network interface to be monitored.

- *IPAddressScope* - An identifier indicating the scope of IP address. For example, GLOBAL, PRIVATE. This identifier is defined in this specification. This identifier is used by a bundle to specify the network interface to be monitored.

Figure 143.1 *Network Interface Information Service Overview Diagram*


Network Interface Information Service bundle
A NetworkAdapter impl
A NetworkAddress impl
<<Interface>> NetworkAdapter
<<Interface>> NetworkAddress
1
0..n
obtain the information of Network Interface
obtain the information of IP address
a Network Interface Information Service user bundle
Bundle using Network Interface Information Service

The NetworkAdapter service provides the network interface information for a logical interface. NetworkAddress service provides the IP address information for an IP address. A NetworkAddress service is associated with a NetworkAdapter service.

When network interface information is changed, the service properties of the corresponing NetworkAdapter service and NetworkAddress service are changed. It is necessary for the bundle using these services to track these services and be advised of changes in the network interface information through Service Events.

143.2 NetworkAdapter Service

NetworkAdapter is an interface that provides information about a single network interface provided by the execution environment. If multiple network interfaces are present, NetworkAdapter services that correspond to each network interface must be registered. NetworkAdapter services must be registered with service properties as shown in the following table.

Table 143.1 *Service properties of NetworkAdapter service*

The key of service property	Type	Description
networkAdapter.type	String	Required property. Network interface type is set to a value.

The key of service property	Type	Description
networkAdapter.hardwareAddress	byte[]	Required property. Hardware address (MAC address) is set to a value. This property can also be obtained from getHardwareAddress().
networkAdapter.name	String	Required property. Network interface name is set to a value. This property can also be obtained from getName().
networkAdapter.displayName	String	Required property. Network interface display name is set to a value. This property can also be obtained from getDisplayName().
networkAdapter.isUp	boolean	Required property. The value is true when a network interface is up and running, otherwise it is false.
networkAdapter.isLoopback	boolean	Required property. The value is true when a network interface is a loopback interface, otherwise it is false.
networkAdapter.isPointToPoint	boolean	Required property. The value is true when a network interface is a point to point interface, otherwise it is false.
networkAdapter.isVirtual	boolean	Required property. The value is true when a network interface is a virtual interface, otherwise it is false.
networkAdapter.supportsMulticast	boolean	Required property. The value is true when a network interface supports multicasting, otherwise it is false.
networkAdapter.parent	String	Required property. Service PID of the NetworkAdapter service which is parent of this NetworkAdapter is specified.
networkAdapter.subInterface	String[]	Required property. Service PID of the NetworkAdapter service which is subinterface of this NetworkAdapter is specified.

When a network interface becomes available, a NetworkAdapter service associated with the network interface is registered with the service registry. If the network interface becomes unavailable, the corresponding NetworkAdapter service is unregistered.

When the attribute values of the network interface change, the NetworkAdapter service is updated with changed service properties. NetworkAdapter interface provides a method corresponding to java.net.NetworkInterface in order to provide information on the associated network interface.

143.2.1 Network Interface Type

Identifying the network interface is possible by using the network interface name. However, since the network interface name is an identifier that is dependent on the operating system, if network interface name is used as identifier, bundles must be implemented to be aware of the operating system. Therefore, in this specification, "network interface type" which is independent of the operating system, is used to identify the network interface. The network interface type string of "LAN" and "WAN" are defined in this specification. This specification allows that Network Interface type other than "LAN"and "WAN" can be defined by the platform provider in each environment. It is provided by the platform provider on which Network Interface Information Service bundle is running. Network Interface type "LAN"indicates the network interface connects to a local area network. Network Interface type "WAN" indicates the network interface connects to an external network (i.e. Internet).

If a bundle wants to obtain the information of the network interface connected to the Internet, the bundle is able to get it by obtaining NetworkAdapter service with the networkAdapter.type service property set to the value "WAN".

This specification allows that Network Interface type other than "LAN"and "WAN" can be defined by the platform provider in each environment. It may be provided by the platform provider on which Network Interface Information Service bundle is running.

Table 143.2 *Network Interface Type*

Network Interface Type	Description
LAN	The network interface to connect to a local area network.
WAN	The network interface to connect to an external network (i.e. Internet).

143.3 NetworkAddress Service

NetworkAddress interface provides information about an IP address available in the execution environment in which the a Network Interface Information Service bundle is running. NetworkAddress service is registered with the service registry together with service properties as shown in the following table.

Table 143.3 *Service properties of NetworkAddress service*

The key of service property	Type	Description
networkAdapter.type	String	Required property. Network interface type is set to a value.
ipAddress.version	String	Required property. IP address version is set to a value.
ipAddress.scope	String	Required property. IP address scope is set to a value.
ipAddress	String	Required property. IP address String is set to a value.
subnetmask.length	int	Required property. Subnet mask length of the required properties IPv4, or IPv6 prefix length is set to a value.
networkAdapter.pid	String	Required property. Service PID of the NetworkAdapterService corresponding to the network interface binding this IP address is set to a value.

A NetworkAddress service is registered with the service registry for each available IP address. When an associated IP address is deleted, or the network interface to which the IP address is bound becomes unavailable, the NetworkAddress service is unregistered. When the associated IP address changes, the NetworkAddress service is updated with updated service properties. A bundle can detect the change of IP address by monitoring the registration or unregistering, updating of the NetworkAddress service. When registering a NetworkAdapter service, the Network Interface Information Service bundle must register it with a unique service PID. Because IP addresses are bound to a network interface, the service PID of the associated NetworkAdapter service and its network interface type are set in the service properties of the NetworkAddress service.

143.3.1 IP Address Version

Defines the version of the IP address. A bundle can select NetworkAddress services using the following IP address version.

Table 143.4 *IP Address Version*

IP Address Version	Description
IPV4	IP address version which means IPv4 address.
IPV6	IP address version which means IPv6 address.

143.3.2 IP address scope

Defins the scope of the IP address. A bundle can select NetworkAddress services using the following IP address scope.

Table 143.5 *IP Address Scope*

IP Address Scope	Description
GLOBAL	IP address scope which means global address.
PRIVATE_USE	IP address scope which means private address.
LOOPBACK	IP address scope which means loopback address.
LINKLOCAL	IP address scope which means link local address.
UNIQUE_LOCAL	IP address scope which means unique-localaddress.
UNSPECIFIED	IP address scope which means the absence of an address.

If a bundle which wants to check for an IP address of the IPv4 global, the bundle is able to confirm by obtaining NetworkAddress service with the ipAddress.version service property set to the value "IPV4" and the ipAddress.scope service property set to the value "GLOBAL".

143.4 A Controller Example

The following example shows the usage of NetworkAddress service. The sample Controller class extends the ServiceTracker class so that it can track NetworkAddress services.

```
class Controller extends ServiceTracker {
  Controller(BundleContext context) {
    super(context, NetworkAdapter.class.getName(), null);
  }

  public Object addingService(ServiceReference ref) {
    NetworkAdapter addAdapter =  (NetworkAdapter)super.addingService(ref);
    String type = addAdapter.getNetworkAdapterType();
    String displayName = addAdapter.getDisplayName();

    // ...

    String servicePID = (String)ref.getProperty(Constants.SERVICE_PID);
    try {
      String filter
         = "(" + NetworkAddress.NETWORKADAPTER_PID + "=" + servicePID + ")";
      ServiceReference[] refs
        = context.getServiceReferences(NetworkAddress.class.getName(), filter);
      for (int i = 0; i < refs.length; i++) {
        NetworkAddress address = (NetworkAddress) context.getService(refs[i]);
        String ipAddress = address.getIpAddress();
        int subnetMaskLength = address.getSubnetMaskLength();
         // ...
       }
    } catch (InvalidSyntaxException e) {
      e.printStackTrace();
    }
    return addAdapter;
  }
}
```

143.5 Security

To acquire network interface information, a bundle needs ServicePermission[NetworkAdapter, GET] and ServicePermission[NetworkAddress, GET]. It can use Filter Based Permissions. When a platform provider performs access control of the bundle, It can set ServicePermission like the following example.

ServicePermission["(&(objectClass=org.osgi.service.networkadapter.NetworkAdapter)(networkAdapter.type=LAN))",GET]

ServicePermission["(&(objectClass=org.osgi.service.networkadapter.NetworkAddress)(networkAdapter.type=LAN) (ipAddress.version=IPV4)(ipAddress.scope=PRIVATE_USE))", GET]

The NetworkAdapter service and the NetworkAddress service should only be implemented by trusted bundles. This bundle requires ServicePermission[NetworkAdapter, REGISTER] and ServicePermission[NetworkAddress, REGISTER].

143.6 org.osgi.service.networkadapter

Network Interface Information Service Specification Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.networkadapter; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.networkadapter; version="[1.0,1.1)"

143.6.1 Summary

- NetworkAdapter - NetworkAdapter is an interface that provides information about single network interfaces provided by the execution environment.
- NetworkAddress - This interface represents an IP address information.

143.6.2 public interface NetworkAdapter

NetworkAdapter is an interface that provides information about single network interfaces provided by the execution environment.

If multiple network interfaces are present, NetworkAdapter Services that correspond to each network interface must be registered. Network interface information service is set the following information as service property.

- NETWORKADAPTER_TYPE : Network Interface Type

- NETWORKADAPTER_DISPLAYNAME : Network Interface Display Name
- NETWORKADAPTER_NAME : Network Interface Name
- NETWORKADAPTER_HARDWAREADDRESS : Hardware Address
- NETWORKADAPTER_IS_UP : Running status of Network Interface
- NETWORKADAPTER_IS_LOOPBACK : To check loopback interface
- NETWORKADAPTER_IS_POINTTOPOINT : To check point to point interface
- NETWORKADAPTER_IS_VIRTUAL : To check virtual interface
- NETWORKADAPTER_SUPPORTS_MULTICAST : To check supports multicasting
- NETWORKADAPTER_PARENT : The PID of parent Network Interface
- NETWORKADAPTER_SUBINTERFACE : The PID of sub Network Interface

When a network interface becomes available, NetworkAdapter service associated with the network interface is registered with the service registry. If the network interface becomes unavailable, the corresponding NetworkAdapter service is unregistered.

When the attribute values of the network interface are set to the service property changes, NetworkAdapter service is updated. NetworkAdapter interface provides a method corresponding to java.net.NetworkInterface in order to provide information on the associated network interface. However, this interface method does not support the Static method. In addition, because NetworkInterface object or InetAddress object is registered in the service registry as NetworkAdapter and NetworkAddress, the NetworkAdapter interface does not provide a method to get those objects. NetworkAdapter provides a method to retrieve the value of an attribute of a network interface.

Concurrency Thread-safe

143.6.2.1 public static final byte[] EMPTY_BYTE_ARRAY

The value byte array of service property, when information is not available.

143.6.2.2 public static final String EMPTY_STRING = ""

The value string of service property, when information is not available.

143.6.2.3 public static final String[] EMPTY_STRING_ARRAY

The value string array of service property, when information is not available.

143.6.2.4 public static final String LAN = "LAN"

The string of network interface type which means the network interface to connect to a local area network.

143.6.2.5 public static final String NETWORKADAPTER_DISPLAYNAME = "networkAdapter.displayName"

The key string of "networkAdapter.displayName" service property.

Network Interface display name is specified. EMPTY_STRING if no display name is available.

143.6.2.6 public static final String NETWORKADAPTER_HARDWAREADDRESS = "networkAdapter.hardwareAddress"

The key string of "networkAdapter.hardwareAddress" service property.

Hardware Address is specified. EMPTY_BYTE_ARRAY if no hardware address is available.

143.6.2.7 public static final String NETWORKADAPTER_IS_LOOPBACK = "networkAdapter.isLoopback"

The key string of "networkAdapter.isLoopback" service property.

The value is true when a network interface is a loopback interface, otherwise it is false.

143.6.2.8 public static final String NETWORKADAPTER_IS_POINTTOPOINT = "networkAdapter.isPointToPoint"

The key string of "networkAdapter.isPointToPoint" service property.

The value is true when a network interface is a point to point interface, otherwise it is false.

143.6.2.9 public static final String NETWORKADAPTER_IS_UP = "networkAdapter.isUp"

The key string of "networkAdapter.isUp" service property.

The value is true when a network interface is up and running, otherwise it is false.

143.6.2.10 public static final String NETWORKADAPTER_IS_VIRTUAL = "networkAdapter.isVirtual"

The key string of "networkAdapter.isVirtual" service property.

The value is true when a network interface is a virtual interface, otherwise it is false.

143.6.2.11 public static final String NETWORKADAPTER_NAME = "networkAdapter.name"

The key string of "networkAdapter.name" service property.

Network Interface Name is specified. EMPTY_STRING if no name is available.

143.6.2.12 public static final String NETWORKADAPTER_PARENT = "networkAdapter.parent"

The key string of "networkAdapter.parent" service property.

Service PID of the NetworkAdapter service which is parent of this NetworkAdapter is specified. EMPTY_STRING if no parent is available.

143.6.2.13 public static final String NETWORKADAPTER_SUBINTERFACE = "networkAdapter.subInterface"

The key string of "networkAdapter.subInterface" service property.

Service PID of the NetworkAdapter service which is subinterface of this NetworkAdapter is specified. EMPTY_STRING_ARRAY if no subinterface is available.

143.6.2.14 public static final String NETWORKADAPTER_SUPPORTS_MULTICAST = "networkAdapter.supportsMulticast"

The key string of "networkAdapter.supportsMulticast" service property.

The value is true when a network interface supports multicasting, otherwise it is false.

143.6.2.15 public static final String NETWORKADAPTER_TYPE = "networkAdapter.type"

The key string of "networkAdapter.type" service property.

Network Interface Type is specified.

143.6.2.16 public static final String WAN = "WAN"

The string of network interface type which means the network interface to connect to an external network (i.e. Internet).

143.6.2.17 public String getDisplayName()

□ Returns the network interface display name of "networkAdapter.displayname" service property value.

Returns Network Interface display name, or null if "networkAdapter.displayname" service property value is empty.

143.6.2.18 public byte[] getHardwareAddress()

□ Returns the MAC address of "networkAdapter.hardwareAddress" service property value.

Returns Hardware Address, or null if "networkAdapter.hardwareAddress" service property value is empty.

143.6.2.19 public int getMTU() throws SocketException

□ Returns the Maximum Transmission Unit (MTU) of this interface.

Returns The value of the MTU for that interface.

Throws SocketException– If an I/O error occurs.

143.6.2.20 public String getName()

□ Returns the network interface name of "networkAdapter.name" service property value.

Returns Network Interface Name, or null if "networkAdapter.name" service property value is empty.

143.6.2.21 public String getNetworkAdapterType()

□ Returns the network interface type of "networkAdapter.type" service property value.

Returns Network Interface Type, or null if "networkAdapter.type" service property value is empty.

143.6.2.22 public boolean isLoopback() throws SocketException

□ Returns whether a network interface is a loopback interface.

Returns true if the interface is a loopback interface.

Throws SocketException– If an I/O error occurs.

143.6.2.23 public boolean isPointToPoint() throws SocketException

□ Returns whether a network interface is a point to point interface.

Returns true if the interface is a point to point interface.

Throws SocketException– If an I/O error occurs.

143.6.2.24 public boolean isUp() throws SocketException

□ Returns whether a network interface is up and running.

Returns true if the interface is up and running.

Throws SocketException– If an I/O error occurs.

143.6.2.25 public boolean isVirtual()

□ Returns whether this interface is a virtual interface (also called subinterface). Virtual interfaces are, on some systems, interfaces created as a child of a physical interface and given different settings (like address or MTU). Usually the name of the interface will the name of the parent followed by a colon (:) and a number identifying the child since there can be several virtual interfaces attached to a single physical interface.

Returns true if this interface is a virtual interface.

143.6.2.26 public boolean supportsMulticast() throws SocketException

□ Returns whether a network interface supports multicasting or not.

Returns true if the interface supports Multicasting.

Throws SocketException– If an I/O error occurs.

143.6.3 public interface NetworkAddress

This interface represents an IP address information.

NetworkAddress interface provides information of IP addresses available in which execution environment on a Network Interface Information Service bundle is running. IP address information service is set the following information as service property.

- NETWORKADAPTER_TYPE : Network Interface Type
- IPADDRESS_VERSION : IP Address Version

- IPADDRESS_SCOPE : IP Address Scope
- IPADDRESS : IP Address
- SUBNETMASK_LENGTH : Subnet Mask Length(IPv4) or Prefix Length(IPv6)
- NETWORKADAPTER_PID : Service PID of the NetworkAdapter service to which this service belongs

NetworkAddress service is registered with the service registry for each available IP address. When associated IP addresses are deleted, or the network interface to which the IP address is bound becomes unavailable, the NetworkAddress service is unregistered. When the associated IP address changes, NetworkAddress service is updated. The user bundle can detect the change of IP address by monitoring the registration or unregistering, updating of NetworkAddress service. Because IP addresses are bound to the network interface, if any, Service PID of the associated NetworkAdapter service and its network interface type are set to service property. NetworkAdapter service MUST be registered after the all associated NetworkAddress services are registered. On the other hand, when unregistering services, after associated NetworkAdapter service is unregistered, NetworkAddress of all related services are unregistered.

Concurrency Thread-safe

143.6.3.1 public static final Integer EMPTY_INTEGER

The value integer of service property, when information is not available.

143.6.3.2 public static final String IPADDRESS = "ipAddress"

The key string of "ipAddress" service property. IP Address is specified.

143.6.3.3 public static final String IPADDRESS_SCOPE = "ipAddress.scope"

The key string of "ipAddress.scope" service property. IP Address scope is specified.

143.6.3.4 public static final String IPADDRESS_SCOPE_GLOBAL = "GLOBAL"

The string of IP address scope which means global address.

The global address is defined as the address other than the address defined in the RFC6890.

143.6.3.5 public static final String IPADDRESS_SCOPE_HOST = "HOST"

The string of IP address scope which means "This host on this network".

See RFC6890 for the definition of "This host on this network".

143.6.3.6 public static final String IPADDRESS_SCOPE_LINKED_SCOPED_UNICAST = "LINKED_SCOPED_UNICAST"

The string of IP address scope which means "Linked-Scoped Unicast".

See RFC6890 for the definition of "Linked-Scoped Unicast".

143.6.3.7 public static final String IPADDRESS_SCOPE_LINKLOCAL = "LINKLOCAL"

The string of IP address scope which means "Link Local".

See RFC6890 for the definition of "Link Local".

143.6.3.8 public static final String IPADDRESS_SCOPE_LOOPBACK = "LOOPBACK"

The string of IP address scope which means "Loopback".

See RFC6890 for the definition of "Loopback".

143.6.3.9 public static final String IPADDRESS_SCOPE_PRIVATE_USE = "PRIVATE_USE"

The string of IP address scope which means "Private-Use Networks".

See RFC6890 for the definition of "Private-Use Networks".

143.6.3.10 public static final String IPADDRESS_SCOPE_SHARED = "SHARED"

The string of IP address scope which means "Shared Address Space".

See RFC6890 for the definition of "Shared Address Space".

143.6.3.11 public static final String IPADDRESS_SCOPE_UNIQUE_LOCAL = "UNIQUE_LOCAL"

The string of IP address scope which means "Unique-Local".

See RFC6890 for the definition of "Unique-Local".

143.6.3.12 public static final String IPADDRESS_SCOPE_UNSPECIFIED = "UNSPECIFIED"

The string of IP address scope which means "Unspecified Address".

See RFC6890 for the definition of "Unspecified Address".

143.6.3.13 public static final String IPADDRESS_VERSION = "ipAddress.version"

The key string of "ipAddress.version" service property. IP Address version is specified.

143.6.3.14 public static final String IPADDRESS_VERSION_4 = "IPV4"

The string of IP address version which means IP address version 4.

143.6.3.15 public static final String IPADDRESS_VERSION_6 = "IPV6"

The string of IP address version which means IP address version 6.

143.6.3.16 public static final String NETWORKADAPTER_PID = "networkAdapter.pid"

The key string of "networkAdapter.pid" service property.

Service PID of the interface information service to which it belongs is specified.

143.6.3.17 public static final String NETWORKADAPTER_TYPE = "networkAdapter.type"

The key string of "networkAdapter.type" service property. Network Interface Type is specified.

143.6.3.18 public static final String SUBNETMASK_LENGTH = "subnetmask.length"

The key string of "subnetmask.length" service property.

Subnet Mask Length(IPv4) or Prefix Length(IPv6) is specified. EMPTY_INTEGER if no length is available.

143.6.3.19 public InetAddress getInetAddress()

□ Returns the InetAddress object of this IP address.

Returned object is created from "ipaddress" service property value.

Returns InetAddress, or null if "ipaddress" service property value is empty.

143.6.3.20 public String getIpAddress()

□ Returns the IP address of "ipaddress" service property value.

Returns IP Address string, or null if "ipaddress" service property value is empty.

143.6.3.21 public String getIpAddressScope()

□ Returns the IP address scope of "ipaddress.scope" service property value.

Returns IP Address Scope, or null if "ipaddress.scope" service property value is empty.

143.6.3.22 public String getIpAddressVersion()

□ Returns the IP address version of "ipaddress.version" service property value.

Returns IP Address Version, or null if "ipaddress.version" service property value is empty.

143.6.3.23 public String getNetworkAdapterPid()

□ Returns the "networkadapter.pid" service property value.

Returns Service ID of the interface information service to which it belongs, or null if "networkadapter.pid" service property value is empty.

143.6.3.24 public String getNetworkAdapterType()

□ Returns the network interface type of "networkAdapter.type" service property value.

Returns Network Interface Type, or null if "networkAdapter.type" service property value is empty.

143.6.3.25 public int getSubnetMaskLength()

□ Returns the "subnetmask.length" service property value.

Returns Subnet Mask Length(IPv4) or Prefix Length(IPv6), or -1 if "subnetmask.length" service property value is empty.

143.7 References

[1] *RFC 6890 : Special-Purpose IP Address Registries*
https://www.ietf.org/rfc/rfc6890.txt, April 2013

144 Resource Monitoring Specification

Version 1.0

144.1 Introduction

Applications, executed on an OSGi platform, need hardware resources (CPU, memory, disk, storage space) and software resources (sockets, threads). As these resources are limited, applications have to share them in order to preserve system quality of service. This is a general fact in OSGi business cases where multiple bundles share the OSGi framework. This is especially the case when the framework is shared by distinct tenants, which are responsible for distinct set of bundles running with their own business logic and lifecycle.

The chapter defines an API for applications to monitor hardware resources consumed by any set of bundles. The bundle is the smallest unit that can be considered as a resource context, the entity that is monitored. Monitored data may enable applications to take decisions on management actions to apply. Resource management actions are mentioned as examples in this chapter, for example, actions on the lifecycle of components, bundles, the framework and the JVM, Java threads, raise of exceptions.

144.2 Essentials

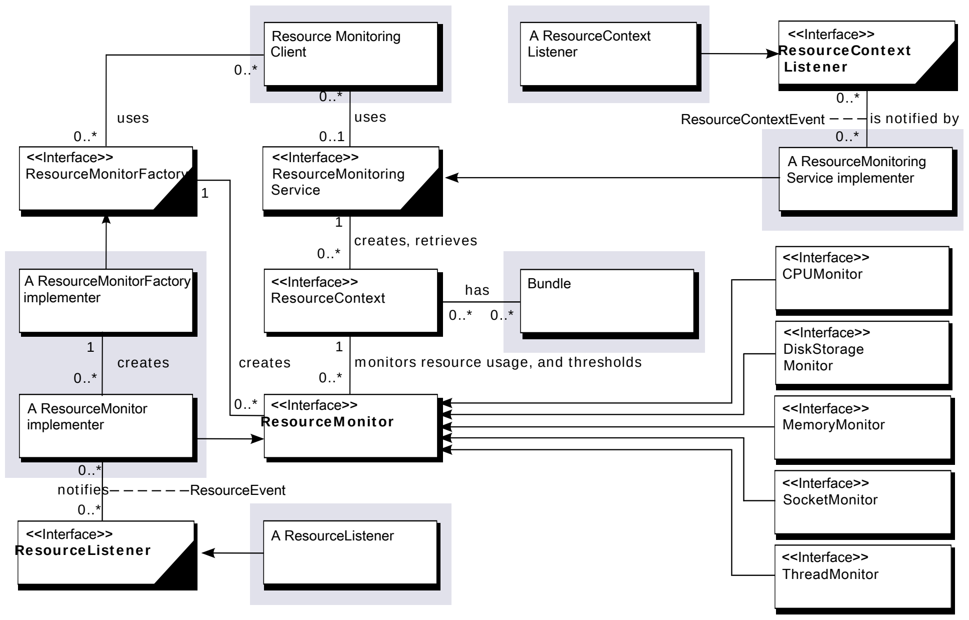
- *Monitoring* - Bundle execution resource usage is monitored.
- *Granular activation* - The resource monitoring service can be activated and deactivated per bundle or per bundle set.
- *Extensibility* - Five resource types are specified (CPU, memory, disk storage, alive thread and in-use sockets). The list of monitored resource types is extensible and query-able.
- *Eventing* - The resource monitoring service notifies interested entities of exceeded limits.

144.3 Entities

- *Resource Context* - A logical entity for resource accounting. A context may be related to a single bundle or a set of bundles.
- *System Resource Context* - Resource context of the core framework.
- *Platform Resource Context* - A Resource context monitoring the resource usage of the platform as a whole.
- *Resource Monitor* - Monitors the usage of a specific resource type for a specific Resource Context. Resource Monitors track resource usage. They hold Resource Thresholds instances. Resource Monitor object implementation may depend on standard or proprietary JVM APIs, and on operating system features.
- *Resource Monitor Factory* - A factory creating Resource Monitor instances for every Resource Context.
- *CPU Monitor* - Resource Monitor used to monitor CPU.
- *Memory Monitor* - Resource Monitor used to monitor memory.

- *Socket Monitor* - Resource Monitor used to monitor socket resource.
- *Disk Storage Monitor* - Resource Monitor for disk storage usage.
- *Thread Monitor* - Resource Monitor used to monitor alive Java Thread objects.
- *Resource Listener* - A Resource Listener receives resource threshold notifications.
- *Resource Event* - A Resource Event defines a notification to be synchronously sent to Resource Listener instances.
- *Resource Context Listener* - A Resource Context Listener receives notifications about resource context creation and configuration.
- *Resource Context Event* - A Resource Context Event defines a notification to be sent to Resource Context Listeners instances.
- *Resource Monitoring Service* - This is a singleton entity which manages Resource Context instances. It is used to create new Resource Context instances and to enumerate existing contexts.
- *Resource Monitoring Client* - Makes any decision to ensure the quality of the service of the system. They use the Resource Monitoring Service to create Resource Context instances. It configures them by adding bundles and Resource Monitors.

Figure 144.1 Resource monitoring class diagram specification.


Resource Monitoring Client
A ResourceContext Listener
<<Interface>> ResourceContext Listener
uses
uses
0..*
0..1
ResourceContextEvent
is notified by
<<Interface>> ResourceMonitorFactory
<<Interface>> ResourceMonitoring Service
A ResourceMonitoring Service implementer
creates, retrieves
A ResourceMonitorFactory implementer
<<Interface>> ResourceContext
has
Bundle
<<Interface>> CPUMonitor
<<Interface>> DiskStorage Monitor
creates
creates
monitors resource usage, and thresholds
A ResourceMonitor implementer
<<Interface>> ResourceMonitor
<<Interface>> MemoryMonitor
<<Interface>> SocketMonitor
notifies
ResourceEvent
<<Interface>> ResourceListener
A ResourceListener
<<Interface>> ThreadMonitor

144.4 Operation Summary

Resource Monitoring Clients use the Resource Monitoring Service service to create Resource Contexts. These clients set bundles or group of bundles to Resource Contexts. They also request every Resource Monitor Factory to create Resource Monitors for a resource type. These Resource Monitors are associated to a single Resource Context.

When activated, Resource Monitors provide the current resource usage per Resource Context. Then, they check whether the current resource usage is compatible with the thresholds held by their associated Resource Listeners. When one of these thresholds is violated, the related Resource Monitor notifies the Resource Listener holding this threshold.

The Resource Monitoring Service manages the set of Resource Contexts. Resource Contexts are persistent between platform restarts. Resource Context Listeners are notified when a Resource Context is created or deleted or when a Resource Context configuration (that is, adding or removing of bundle) is updated.

144.5 Resource Context

A ResourceContext instance is a logical entity used to account resource usage. Every Resource Context defines a bundle scope which can be either a single bundle or a set of bundles. Once the bundle scope is defined, resources used by those bundles are monitored through a set of per-resource-type Resource Monitor instances.

Resource Context instances are persistent. The persistence of those instances is directly managed by the Resource Monitoring Service instance.

Each Resource Context is uniquely identified by a name. It can be retrieved through the getName() method. It can not be changed, that is it is definitively set when the Resource Context instance is created.

The Resource Context bundle scope is retrieved through the getBundleIds() method. This bundle scope can be extended through the addBundle(long) method. Bundles can also be removed from a Resource Context through the removeBundle(long,ResourceContext) method. For this last method, a Resource Context instance MAY be specified in order to associate the removed bundle to another Resource Context instance.

Resource Monitor instances are retrieved through getMonitor(String) method or the getMonitors() method. The list of available resource types is retrieved through the Resource Monitoring Service singleton instance.

Resource Monitor instances are added to and removed from a Resource Context instance by calling either addResourceMonitor(ResourceMonitor) method or removeResourceMonitor(ResourceMonitor) method. Both methods SHOULD only be called by ResourceMonitorFactory instances (see createResourceMonitor(ResourceContext) method).

A Resource Context is retrieved through the Resource Monitoring Service service.

A Resource Context instance can be deleted through removeContext(ResourceContext) method. The Resource Context input argument then defines a destination Resource Context instance for the bundles belonging to the to-be-removed Resource Context instance.

144.6 System Resource Context

The System Resource Context is the Resource Context of the execution environment for the running OSGi bundles. It includes the resources of bundle "0". It is retrieved through the Resource Monitoring Service service.

The name of this context is “system”. See SYSTEM_CONTEXT_NAME.

144.7 Framework Resource Context

The Framework Resource Context is a Resource Context monitoring resources of the platform as a whole. It is retrieved through the Resource Monitoring Service service. This Resource Context holds all hosted bundles allowing access to the whole platform resource consumption.

The name of this context is “framework”. See FRAMEWORK_CONTEXT_NAME.

144.8 Resource Monitor

A ResourceMonitor instance monitors a resource type consumed by the bundles of a specific Resource Context instance.

A Resource Context instance holds at most one Resource Monitor instance per monitor-able resource type. Resource Monitor instances are retrieved through their related Resource Context instance. Resource Monitor instances give access to their related Resource Context instance through a call to See getContext() method.

The monitored resource type is retrieved through the getResourceType() method.

The current usage of a resource consumed by a Resource Context instance is given through the getUsage() method. This method returns a Java Object to be casted to the appropriate Java object type depending on the Resource type. The next table provides the expected Java Object type for each specified resource type:

Table 144.1 *Table of resource types.*

Type of Resource	Expected Java Object type	Value description
CPU	Long	Cumulative CPU time in ns.
Memory	Long	Allocated memory in bytes.
Threads	Long	Number of alive thread.
Socket	Long	Number of in-use socket.
Disk storage space	Long	**Bytes on the bundle persistent storage area.**

For example, for a MemoryMonitor instance, a call to getUsage() returns a Long java object indicating the amount of memory the related Resource Context instance is consuming.

A Resource Monitor instance is enabled and disabled through enable() and disable() methods. The state (enabled or disabled) of a Resource Monitor is retrieved through a call to isEnabled() method. Enable and disable monitoring mechanisms on-the-fly on localized set of bundles may be crucial for performance issues. See [1] *Adaptive Monitoring of End-user OSGi based Home Boxes.*

A Resource Monitor instance can also be deleted (delete() method). isDeleted() method returns true if the ResourceMonitor instance has been deleted.

Five types of Resource Monitor are specified:

- CPU Monitor
- Memory Monitor
- Socket Monitor
- Disk Storage Monitor
- Thread Monitor

The support of any Resource Monitor is optional. This list MAY be extended by the solution vendor. The list of the types that are supported on the OSGi platform can be computed by querying ResourceMonitorFactory services. Resource monitoring algorithms may vary with factories, see [2] *Memory Monitoring in a Multi-tenant OSGi Execution Environment.* They are out of the scope of this specification.

144.9 Resource Monitor Factory

A ResourceMonitorFactory is a service that provides Resource Monitor instances of a specific resource type (for example, CPUMonitor, MemoryMonitor, etc.) for every Resource Context.

Every Resource Monitor Factory service is registered with the org.osgi.resourcemonitoring.ResourceType mandatory property, see RESOURCE_TYPE_PROPERTY. This property indicates which type of Resource Monitor a Resource Monitor Factory is able to create. The type can also be retrieved through a call to getType(). The type MUST be unique (two Resource Monitor Factory services MUST not have the same type).

New Resource Monitor instances are created by a call to createResourceMonitor(ResourceContext). This method returns a new Resource Monitor instance associated to the provided Resource Context instance. The ResourceMonitorFactory MUST call addResourceMonitor(ResourceMonitor) to associate the newly created ResourceMonitor with the provided ResourceContext instance. The newly created Resource Monitor is disabled, that is, it is initially not monitoring the Resource Context resource consumption. It can be activated through a call to enable().

Resource Monitor instances are deleted by calling delete() method.

A Resource Monitor instance MUST only be created through its ResourceMonitorFactory.

Resource Monitor Factory instances should be only used by the Resource Monitoring Service singleton instance. The Resource Monitoring Service singleton instance performs a service lookup on all existing Resource Monitor Factories. It uses a Resource Monitor Factory instance when it has to create a new Resource Context instance and their associated Resource Monitor instances.

144.10 CPU Monitor

A CPUMonitor instance is a Resource Monitor used to monitor the CPU usage of the bundles belonging to a Resource Context.

CPU usage and thresholds are expressed as a cumulative number of nanoseconds (long). The encapsulated value can be retrieved with the getCPUUsage() method.

In case where a threshold is reached, the CPU Monitor instance generates an event triggering Resource Monitoring Clients defined corrective actions (for example, decrease thread priority).

144.11 Memory Monitor

A MemoryMonitor instance monitors and limits the memory used by the bundles of a Resource Context instance.

Memory is accounted as bytes. Memory usage and thresholds are long java objects. The encapsulated value can be retrieved through the getMemoryUsage() method.

When an error threshold is reached, the next memory allocation MAY be prevented by the system and MAY throw a specific Exception in the associated context.

144.12 Socket Monitor

A SocketMonitor instance monitors and limits the number of existing sockets (for example, TCP, UDP) which are considered to be in use (for example, listening for incoming packet, bound, or sending outgoing packets).

A Socket is considered to be in-use state when a native socket file descriptor is created. It leaves this state when this socket file descriptor is deleted.

The number of in-use sockets is a long. The encapsulated value can be retrieved using getSocketUsage() method.

When an ERROR threshold is reached, the next socket file descriptor creation in the associated context MAY throw a SocketException.

144.13 Disk Storage Monitor

A DiskStorageMonitor instance monitors and limits the use of persistent storage within Bundle Persistent Storage Area a Resource Context (the bundles actually belonging to it) consumes.

Disk Storage is expressed as a number of bytes of type long. The encapsulated value can be retrieved using getUsedDiskStorage() method.

An IOException MAY be thrown in the associated context when an ERROR threshold is reached.

144.14 Thread Monitor

A ThreadMonitor instance monitors and limits the number of alive Java Thread objects for a Resource Context instance. A Thread is considered to be alive when it is in the RUNNABLE, BLOCKED, WAITING or TIMED_WAITING thread state.

Usage and thresholds are Java int objects. The encapsulated value can be retrieved using getAliveThreads() method.

When an ERROR threshold is reached, any further thread activation will be prevented in the associated context. An InternalError exception MAY also be thrown in the associated context.

144.15 Resource Listener

A ResourceListener receives notifications about resource usage for a specific Resource Context and a specific type of resource. A notification will be sent to a Resource Listener when one of its thresholds is violated.

A Resource Listener holds two types of threshold:

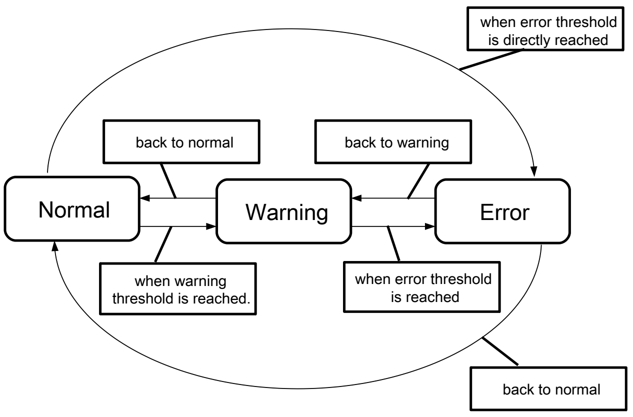
- A lower threshold type. This kind of threshold is reached when the monitored resource usage decreases below the threshold.
- An upper threshold type. An upper threshold is reached when the monitored resource usage exceeds this threshold.

Each of them have two levels:

- a WARNING level.
- an ERROR level.

A threshold has the following state diagram, which transitions are associated to events:

Figure 144.2 Threshold state diagram.


when error threshold is directly reached
back to normal
back to warning
Normal
Warning
Error
when warning threshold is reached.
when error threshold is reached
back to normal

A threshold state depends on the current consumption of resource and the type of threshold (upper or lower threshold).

A Resource Listener is registered as an OSGi service. The implementer must provide the two following mandatory properties:

- RESOURCE_CONTEXT property – a String defining the name of Resource Context for which the Listener want to receive threshold notifications.
- RESOURCE_TYPE property – a String defining which type of resource the listener wants to monitor.

It also has to provide at least one of these four properties when registered as an OSGi service:

- UPPER_WARNING_THRESHOLD
- UPPER_ERROR_THRESHOLD
- LOWER_WARNING_THRESHOLD
- LOWER_ERROR_THRESHOLD

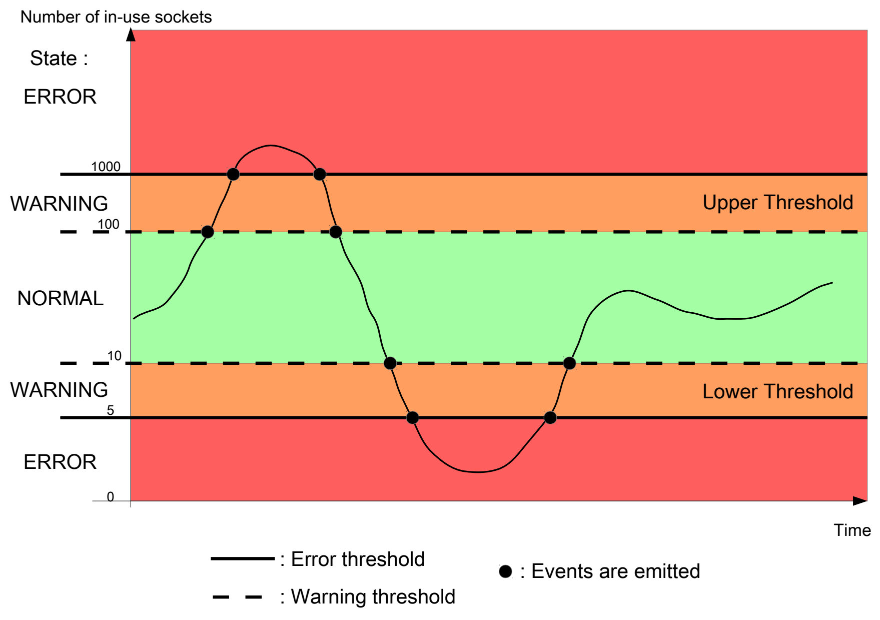
These properties are mapped to the four types of threshold values a Resource Listener may support. The service properties are used to notify the associated Resource Monitor when one of these threshold values is modified.

Threshold values can also be retrieved through a set of getter methods. All of these methods returns a Comparable object used by the associated Resource Monitor in order to determine the current state of the current usage.

RESOURCE_CONTEXT and RESOURCE_TYPE properties are used by Resource Monitors to identify their associated Resource Listeners. Once associated, a Resource Monitor retrieves the threshold settings using service properties. When one of its thresholds is reached, the Resource Monitor calls notify(ResourceEvent).

Two examples of resource consumption are explained below, first with in-use sockets monitoring, second with CPU monitoring. The next picture shows the state diagram of the number of in-use state socket over the time.

Figure 144.3 *Number of in-use sockets over the time.*



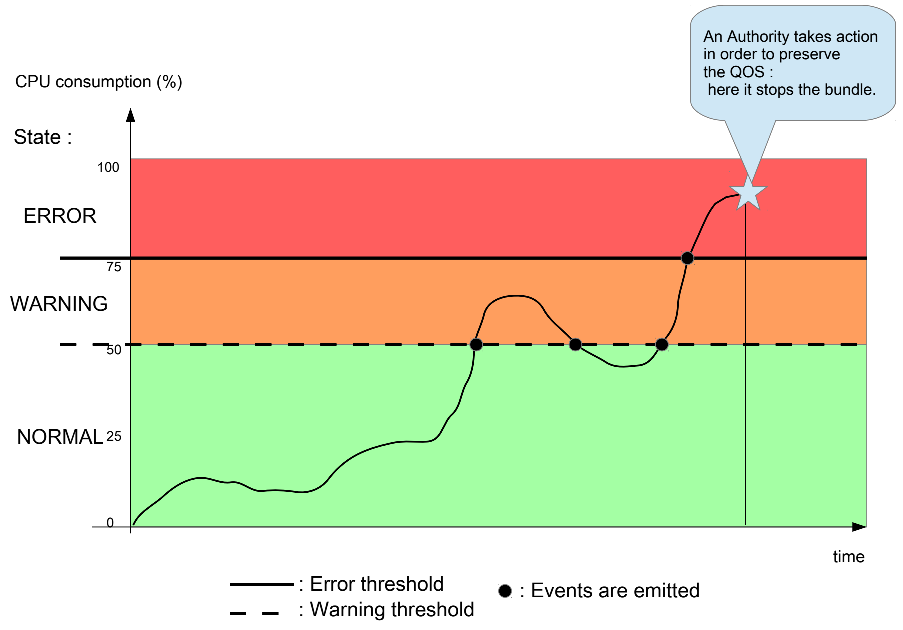
In our example, the lower warning threshold and the lower error threshold of the Resource Listener are respectively set to 10 and 5. When the number of in-use sockets decreases under 10, the usage goes from the NORMAL state to the WARNING state and the Resource Listener receives a WARNING event. If the number of in-use state sockets decreases again and goes down to 5, the usage goes from the WARNING state to the ERROR state and the Resource Listener receives a ERROR Resource Event.

The upper threshold is also set. The upper warning threshold and the upper error threshold are respectively set to 100 and 1000 in-use state sockets. When the number of sockets reaches 100, the usage goes from the NORMAL state to the WARNING state and the Resource Listener receives a WARNING Resource Event. If this number is still increasing and exceeds 1000, then the usage goes from the WARNING state to the ERROR state and the Resource Listener receives an ERROR Resource Event.

This is a typical use case for a Java Web server. Indeed, one of the most important quality of service indicator is the number of in-use state sockets a java web server is handling. A low number of in-use state sockets may indicate the java web server encounters network problems. On the contrary, a high number of in-use state socket may be the result of an external network attack or it could also indicates the java web server is overused and its administrator should take actions to load-balance the charge to another java web server instance.

For other resource types, only upper thresholds may be useful. The next diagram shows the CPU consumption a Resource Context is using over the time:

Figure 144.4 CPU consumption (%) over the time – Upper Threshold.



In this example, only the upper threshold is set. The upper warning threshold is set to 50%, the error one is set to 75%. CPU consumption fluctuates between 0 and 50%, the usage is in the NORMAL state. Then it increases and reaches 50%. The usage then goes from the NORMAL state to the WARNING state and the Resource Listener holding the threshold receives a WARNING Resource Event.

Afterwards, CPU consumption decreases under 50%; the usage goes from the WARNING state to the NORMAL state. The related Resource listener receives a NORMAL Resource Event.

It then increases again and exceeds 50%. The usage goes to the WARNING state. CPU consumption is still increasing and exceeds 75%. At this moment, the usage goes from the WARNING state to the ERROR state and the related Resource Listener receives an ERROR Resource Event.

After some seconds in the ERROR state, the Resource Listener implementation stops the bundle in order to preserve the quality of service.

The choice of the type of threshold (lower or upper, or both of them) depends on the type of resource and the needs of the Resource Monitoring Clients providing the Resource Listener. Other resources like the free memory may take advantage of a lower threshold.

144.16 Resource Event

A ResourceEvent instance is an event synchronously sent to a Resource Listener when one of its thresholds is reached. This event is notified to a Resource Listener through a call to ResourceListener.notify(ResourceEvent).

A Resource Event has a type among the following ones:

- ERROR – The resource consumption reaches either the upper or the lower error threshold of the Resource Listener receiving this event.

- WARNING – The resource consumption reaches either the upper or the lower warning threshold of the Resource Listener receiving this event.
- NORMAL – The resource consumption is back from warning or error state to normal state.

The Resource Listener instance analyzes this event by calling the following methods:

- getValue() method returns the resource consumption at the time when the Resource Event instance was generated.
- isUpperThreshold() method returns true if the reached threshold is an upper threshold type. If this method returns false, this is a lower threshold.
- getType() method indicates the state (WARNING, ERROR, or NORMAL) of the resource usage.
- getContext() method returns the Resource Context instance related to this event. The Resource Listener can use it to retrieve the Resource Monitor instance. For example, event.getContext().getMonitor(event.getResourceType()).

144.17 Resource Context Listener

A ResourceContextListener instance receives notifications about Resource Context lifecycle and configuration.

A notification will be sent when:

- A Resource Context is created.
- A Resource Context is updated, that is, a bundle has been added or removed from a Resource Context instance.
- A Resource Context is deleted.

An application which is interested in notifications has to register a Resource Context Listener instance as an OSGi service. The application may provide a set of properties at registration time to reduce the number of notifications a Resource Listener instance will receive. The available property is:

- RESOURCE_CONTEXT property – An array of String defining the name of Resource Context instances. If defined, a Resource Listener instance will only receive notifications related to these specified Resource Context instances.

A Resource Context Listener instance is notified through a call to notify(ResourceContextEvent) method.

144.18 Resource Context Event

A ResourceContextEvent instance is an event sent to Resource Context Listener instances through a call to the notify(ResourceContextEvent) method.

A Resource Context Event has a type among the four following ones:

- RESOURCE_CONTEXT_CREATED – A new Resource Context instance has been created.
- RESOURCE_CONTEXT_REMOVED – A Resource Context instance has been deleted.
- BUNDLE_ADDED – A bundle has been added in the scope of a Resource Context instance.
- BUNDLE_REMOVED – A bundle has been removed from the scope of a Resource Context instance.

In the case of a RESOURCE_CONTEXT_CREATED event or a RESOURCE_CONTEXT_REMOVED event, a call to getContext() returns the targeted Resource Context instance.

In the case of a BUNDLE_ADDED type or BUNDLE_REMOVED type, getBundleId() returns the id of the bundle to be added to or removed from. The related Resource Context instance is given by a call to getContext().

144.19 Resource Monitoring Service

The ResourceMonitoringService manages the Resource Context instances. The Resource Monitoring Service is available through the OSGi service registry.

This service holds the existing Resource Context instances. Resource Context instances are created by calling the createContext(String,ResourceContext) method. The caller provides a context name as a string and optionally a template as a ResourceContext object.

The list of existing Resource Context instances can be retrieved through the following methods:

- getContext(String) – returns the ResourceContext with the specified resource context name.
- getContext(long) – returns the ResourceContext associated to the provided bundle id.
- listContext() – retrieve all existing Resource Context instances as an array.

The Resource Monitoring Service singleton manages the persistence of the Resource Context instances. The following properties are stored:

- name of the Resource Context.
- list of the bundles belonging to the Resource Context.
- list of the Resource Monitor instances. For each one: the sampling period, and the monitoring period.

The way the Resource Monitoring Service persists the Resource Context instances is implementation specific. The implementer is free to use any file format and file location it wants. At startup, the Resource Monitoring Service will load the persisted Resource Context instances to restore the state prior to shutdown.

144.20 Resource Monitoring Client

A Resource Monitoring Client uses the Resource Monitoring Service singleton instance to apply Resource Monitoring policies. These entities MAY:

- create and configure Resource Context instances (resource thresholds, bundle scope)
- take any decisions (stop a bundle, uninstall a bundle) if a Resource Context exceeds resource limit.

These policies are out of the scope of this specification.

144.21 Security

It is recommended that ServicePermission[ResourceMonitoringService|ResourceMonitoringFactory|ResourceListener, REGISTER|GET] be used sparingly and only for bundles that are trusted.

144.22 org.osgi.service.resourcemonitoring

Resource Monitoring Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.resourcemonitoring; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.resourcemonitoring; version="[1.0,1.1)"
```

144.22.1 Summary

- `ResourceContext` - Logical entity for resource accounting.
- `ResourceContextEvent` - A Resource Context Event instance is an event sent to Resource Context Listener instances through a call to ResourceContextListener.notify(ResourceContextEvent) method.
- `ResourceContextException` - Resource Context Exception.
- `ResourceContextListener` - A `ResourceContextListener` is notified whenever:
  - a ResourceContext is created or deleted.
  - a bundle is added or removed from a ResourceContext.
- `ResourceEvent` - An event is sent to a ResourceListener when resource usage violates one of their thresholds.
- `ResourceListener` - A ResourceListener is an OSGi service which is notified when a Resource Context violates one of the threshold defined by the listener.
- `ResourceMonitor` - Representation of the state of a resource for a resource context.
- `ResourceMonitorException` - Resource Monitor Exception reports an invalid usage of a monitor.
- `ResourceMonitorFactory` - A Resource Monitor Factory is a service that provides Resource Monitor instances of a specific resource type (for example, CPUMonitor, MemoryMonitor...) for every Resource Context.
- `ResourceMonitoringService` - It manages the Resource Context instances.

144.22.2 public interface ResourceContext

Logical entity for resource accounting. A resource context has a group of member bundles, and a bundle can be a member of 0 or 1 resource context.

Resource Monitoring Clients can use the ResourceMonitoringService.createContext(String, ResourceContext) method to create ResourceContext instances.

Resource Monitoring Clients can use the getMonitor(String) method to get ResourceMonitor instances for the supported resource types. These instances can then be used to monitor the usage of the resources, or the set usage limits.

ResourceContexts are retrieved through the ResourceMonitoringService OSGi service.

144.22.2.1 public void addBundle(long bundleId) throws ResourceContextException

bundleId The bundle to add to this resource context

□ Adds a bundle to the resource context. The bundle will be a member of the context until it is uninstalled, or explicitly removed from the context with removeBundle(long) method or removeBundle(long, ResourceContext) method.

Resources previously allocated by this bundle (in another resource context) will not be moved to this resource context. The change applies only for future allocations.

A ResourceContextEvent with type ResourceContextEvent.BUNDLE_ADDED will be sent.

Throws `ResourceContextException` – For example, when the bundle can't be added to the ResourceContext.

144.22.2.2 public void addResourceMonitor(ResourceMonitor<?> resourceMonitor) throws ResourceContextException

resourceMonitor resourceMonitor instance to be added

□ Adds a new ResourceMonitor instance monitoring resource for this resource context. This method should be called only by ResourceMonitorFactory instance.

Throws `ResourceContextException` – For example, when the monitor can't be added to the ResourceContext.

144.22.2.3 public boolean equals(Object resourceContext)

resourceContext resource context

□ A ResourceContext rc1 is equals to ResourceContext rc2 if rc1.getName() is equals to rc2.getName().

Returns true if getName().equals(resourceContext.getName()

144.22.2.4 public long[] getBundleIds()

□ Returns the bundle identifiers belonging to this Resource Context.

Returns An array of Bundle objects, or an empty array if no bundles are currently members of this context

144.22.2.5 public ResourceMonitor<?> getMonitor(String resourceType) throws ResourceContextException

resourceType The resource type, for which a resource monitor is requested

□ Returns a ResourceMonitor instance for the specified resource type. If the ResourceMonitoringService implementation does not support this resource type, null is returned

Returns A ResourceMonitor instance, or null, if this resource type is not supported

Throws `ResourceContextException` – For example, when the monitor(s) can't be retrieved from the ResourceContext.

144.22.2.6 public ResourceMonitor<?>[] getMonitors() throws ResourceContextException

□ Retrieves all the existing ResourceMonitor belonging to this context.

Returns an array of ResourceMonitor. May be empty if no ResourceMonitor

Throws `ResourceContextException` – For example, when the monitor(s) can't be retrieved from the ResourceContext.

144.22.2.7 public String getName()

□ Returns the name of the resource context. Resource context names are unique within a framework instance.

Returns The resource context name

144.22.2.8 public int hashCode()

□ Retrieves the hashCode value of a ResourceContext. The hashCode value of a ResourceContext is only based on the hashcode value of the name of the context.

Returns hashcode

144.22.2.9 public void removeBundle(long bundleId) throws ResourceContextException

bundleId bundle identifier

□ Removes the bundle identified by bundleId from the Resource Context. The bundle is no longer to this Resource Context.

Throws `ResourceContextException`– For example, when the bundle can't be removed from the ResourceContext.

144.22.2.10 public void removeBundle(long bundleId, ResourceContext destination) throws ResourceContextException

bundleId the identifier of the bundle to be removed from the Resource Context

destination A resource context in which to add the bundle, after removing it from this context. If no destination is provided (that is null), the bundle is not associated to a new Resource Context.

□ Removes the bundle from this resource context. If a `destination` context is specified, the bundle will be added in it.

Resources previously allocated by this bundle will not be removed from the resource context. The change applies only for future allocations.

A ResourceContextEvent with type ResourceContextEvent.BUNDLE_REMOVED will be sent.

Throws `ResourceContextException`– For example, when the bundle can't be removed from the ResourceContext.

144.22.2.11 public void removeContext(ResourceContext destination) throws ResourceContextException

destination The ResourceContext where the resources currently allocated by this resource context will be moved.

□ Removes a resource context. All resources allocated in this resource context will be moved to the `destination` context. If `destination` is `null`, these resources will no longer be monitored.

A ResourceContextEvent with type ResourceContextEvent.RESOURCE_CONTEXT_REMOVED will be sent.

Throws `ResourceContextException`– For example, when the resource context can't be removed.

144.22.2.12 public void removeResourceMonitor(ResourceMonitor<?> resourceMonitor) throws ResourceContextException

resourceMonitor resource monitor instance to be removed

□ Removes a ResourceMonitor instance from the context.

Throws `ResourceContextException`– For example, when the monitor can't be removed from the ResourceContext.

144.22.3 public class ResourceContextEvent

A Resource Context Event instance is an event sent to Resource Context Listener instances through a call to ResourceContextListener.notify(ResourceContextEvent) method. A Resource Context Event has a type among the four following ones:

- RESOURCE_CONTEXT_CREATED – A new Resource Context instance has been created.
- RESOURCE_CONTEXT_REMOVED – A Resource Context instance has been deleted.
- BUNDLE_ADDED – A bundle has been added in the scope of a Resource Context instance.
- BUNDLE_REMOVED – A bundle has been removed from the scope of a Resource Context instance.

144.22.3.1 public static final int BUNDLE_ADDED = 2

A bundle has been added to e ResourceContext

The ResourceContext.addBundle(long) method has been invoked

144.22.3.2 public static final int BUNDLE_REMOVED = 3

A bundle has been removed from a ResourceContext

ResourceContext.removeBundle(long) method or ResourceContext.removeBundle(long, Resource-Context) method have been invoked, or the bundle has been uninstalled

144.22.3.3 **public static final int RESOURCE_CONTEXT_CREATED = 0**

A new ResourceContext has been created.

The ResourceMonitoringService.createContext(String, ResourceContext) method has been invoked.

144.22.3.4 **public static final int RESOURCE_CONTEXT_REMOVED = 1**

A ResourceContext has been removed

The ResourceContext.removeContext(ResourceContext) method has been invoked

144.22.3.5 **public ResourceContextEvent(int pType, ResourceContext pResourceContext)**

pType event type

pResourceContext context

□ Create a new ResourceContextEvent. This constructor should be used when the type of the event is either RESOURCE_CONTEXT_CREATED or RESOURCE_CONTEXT_REMOVED.

144.22.3.6 **public ResourceContextEvent(int pType, ResourceContext pResourceContext, long pBundleId)**

pType event type

pResourceContext context

pBundleId bundle

□ Create a new ResourceContextEvent. This constructor should be used when the type of the event is either BUNDLE_ADDED or BUNDLE_REMOVED.

144.22.3.7 **public boolean equals(Object varo)**

144.22.3.8 **public long getBundleId()**

Retrieves the identifier of the bundle being added to or removed from the Resource Context.

This method returns a valid value only when getType() returns:

- BUNDLE_ADDED
- BUNDLE_REMOVED

Returns the bundle id or -1 (invalid value).

144.22.3.9 **public ResourceContext getContext()**

□ Retrieves the Resource Context associated to this event

Returns Resource Context.

144.22.3.10 **public int getType()**

□ Retrieves the type of this Resource Context Event.

Returns the type of the event. One of:

- RESOURCE_CONTEXT_CREATED
- RESOURCE_CONTEXT_REMOVED
- BUNDLE_ADDED
- BUNDLE_REMOVED

144.22.3.11 public int hashCode()

144.22.3.12 public String toString()

144.22.4 public class ResourceContextException extends Exception

Resource Context Exception.

144.22.4.1 public ResourceContextException(String msg)

msg message

□ Create a new ResourceContextException

144.22.4.2 public ResourceContextException(String msg, Throwable t)

msg message

t exception

□ Create a new ResourceContextException

144.22.5 public interface ResourceContextListener

A ResourceContextListener is notified whenever:

- a ResourceContext is created or deleted.
- a bundle is added or removed from a ResourceContext.

A ResourceContextListener is registered as an OSGi service. At registration time, the following property may be provided:

- the RESOURCE_CONTEXT property which limits the Resource Context for which notifications will be received. This property can be either a String value or an array of String. If this property is not set, the Resource Context Listener receives events from all the Resource Context.

144.22.5.1 public static final String RESOURCE_CONTEXT = "resource.context"

Property specifying the ResourceContext(s) for which a notification will be received by this listener.

The property value is either a string (i.e the name of the ResourceContext) and an array of string (several ResourceContext).

144.22.5.2 public void notify(ResourceContextEvent event)

event event.

□ Notify this listener about a ResourceContext events.

144.22.6 public class ResourceEvent<T>

<T> The type for the Resource.

An event is sent to a ResourceListener when resource usage violates one of their thresholds.

ResourceEvent objects are delivered synchronously to all matching ResourceListener services. A typed code is used to identify the event.

See Also ResourceListener

144.22.6.1 public static final int ERROR = 2

Type of ResourceEvent indicating a threshold goes to the ERROR state.

144.22.6.2 public static final int NORMAL = 0

Type of ResourceEvent indicating a threshold goes to the NORMAL state.

144.22.6.3 public static final int WARNING = 1

Type of ResourceEvent indicating a threshold goes to the WARNING state.

144.22.6.4 public ResourceEvent(int pType, ResourceContext pContext, boolean pIsUpperThreshold, Comparable<T> pValue)

pType the event type

pContext the resource context

pIsUpperThreshold whether it is an upper threshold

pValue the value

□ Creates a new ResourceEvent.

144.22.6.5 public boolean equals(Object varo)

144.22.6.6 public ResourceContext getContext()

□ Returns the resource context that caused the event.

Returns The resource context that caused the event.

144.22.6.7 public int getType()

□ Returns the event type. The type values are:

- NORMAL
- WARNING
- ERROR

Returns The event type

144.22.6.8 public Comparable<T> getValue()

□ Returns the resource consumption value. Relevant only for event types NORMAL, WARNING and ERROR.

Returns the resource consumption value, or null if a resource monitor is not relevant.

144.22.6.9 public int hashCode()

144.22.6.10 public boolean isUpperThreshold()

□ Returns true if the threshold triggering this event is an upper threshold. This method is only used when getType() returns NORMAL, WARNING or ERROR.

Returns true if it is an upper threshold.

144.22.6.11 public String toString()

144.22.7 public interface ResourceListener<T>

<T> The type for the Resource.

A ResourceListener is an OSGi service which is notified when a Resource Context violates one of the threshold defined by the listener.

Every ResourceListener is associated to a specific Resource Context and a specific Resource type. It defines two types of thresholds: a lower and a upper. A lower threshold is reached when the resource usage decreases below the threshold. On the contrary, an upper threshold is reached when the resource usage exceeds the threshold.

Both lower or upper threshold are two levels: a warning level and error level. The warning level indicates the resource usage becomes to be critical but are still acceptable. The error level indicates the resource usage is now critical for the overall system and actions should be taken.

A Resource Listener is registered with these two mandatory properties:

- RESOURCE_CONTEXT which defines the ResourceContext associated to this Listener
- RESOURCE_TYPE which the type of resource

The next optional properties are used to specify threshold values. A ResourceListener must at least provides one of them:

- ResourceListener.UPPER_WARNING_THRESHOLD
- ResourceListener.UPPER_ERROR_THRESHOLD
- ResourceListener.LOWER_WARNING_THRESHOLD
- ResourceListener.LOWER_ERROR_THRESHOLD

These threshold values can also be retrieved through methods.

Resource Listeners are associated to a Resource Context and a Resource Monitor based on the RESOURCE_CONTEXT property and the RESOURCE_TYPE property (both of them are mandatory at registration time).

Once associated, the ResourceMonitor gets the threshold values through the service properties (i.e UPPER_WARNING_THRESHOLD, UPPER_ERROR_THRESHOLD, LOWER_WARNING_THRESHOLD and LOWER_WARNING_THRESHOLD) and store them. Once it detects a new resource consumption, it compares the new resource usage value with the thresholds provided by the Resource Listener. If the resource usage violates one of these thresholds, the Resource Monitor notifies the ResourceListener through a call to notify(ResourceEvent).

A ResourceMonitor tracks threshold value modification by using a ServiceListener.

144.22.7.1 public static final String LOWER_ERROR_THRESHOLD = "lower.error.threshold"

Optional property defining the value of the lower error threshold.

144.22.7.2 public static final String LOWER_WARNING_THRESHOLD = "lower.warning.threshold"

Optional property defining the value of the lower warning threshold.

144.22.7.3 public static final String RESOURCE_CONTEXT = "resource.context"

Mandatory property specifying the Resource Context associated with the listener.

144.22.7.4 public static final String RESOURCE_TYPE = "resource.type"

Mandatory property defining the type of Resource (i.e the ResourceMonitor) associated to this Listener.

144.22.7.5 public static final String UPPER_ERROR_THRESHOLD = "upper.error.threshold"

Optional property defining the value of the upper error threshold.

144.22.7.6 public static final String UPPER_WARNING_THRESHOLD = "upper.warning.threshold"

Optional property defining the value of the upper warning threshold.

144.22.7.7 public Comparable<T> getLowerErrorThreshold()

□ Retrieves the lower error threshold value set by the listener. If the resource usage decreases under this threshold, the notify(ResourceEvent) will be called. The provided ResourceEvent then indicates the ERROR state is reached.

Returns a comparable object or null if no threshold is set.

144.22.7.8 public Comparable<T> getLowerWarningThreshold()

□ Retrieves the lower warning threshold value set by the listener. If the resource usage decreases under this threshold value, the notify(ResourceEvent) will be called. The provided ResourceEvent then indicates the WARNING state is reached.

Returns a comparable object or null if no threshold is set.

144.22.7.9 public Comparable<T> getUpperErrorThreshold()

□ Retrieves the upper error threshold value set by this listener. If the resource usage exceeds this threshold, the notify(ResourceEvent) will be called. The provided ResourceEvent then indicates the ERROR state is reached.

Returns a comparable object or null if no threshold is reached.

144.22.7.10 public Comparable<T> getUpperWarningThreshold()

□ Retrieves the upper warning threshold value set by this listener. If the resource usage exceeds this threshold, the notify(ResourceEvent) method will be called. The provided ResourceEvent then indicates the WARNING state is reached.

Returns a comparable object or null if no threshold is reached.

144.22.7.11 public void notify(ResourceEvent<T> event)

event The ResourceEvent object

□ Receives a resource monitoring notification

144.22.8 public interface ResourceMonitor<T>

<T> The type for the Resource.

Representation of the state of a resource for a resource context.

ResourceMonitor objects are returned by the ResourceContext.getMonitor(String) method.

The `ResourceMonitor` object may be used to:

- Enable/Disable the monitoring of the corresponding resource type for the corresponding resource context
- View the current usage of the resource by this resource context

A resource monitor can have a sampling period, a monitored period, or both. For example, for CPU monitoring, the resource monitor implementation can get the CPU usage of the running threads once per minute, and calculate the CPU usage per context in percentages based on the last ten such measurements. This could make a 60 000 milliseconds sampling period, and a 600 000 milliseconds monitored period.

144.22.8.1 public void delete() throws ResourceMonitorException

□ Disable and delete this instance of Resource Monitor. This method MUST update the list of ResourceMonitor instances hold by the Resource Context (getContext().removeMonitor(this)).

Throws `ResourceMonitorException`– For example, when the monitor can't be removed from the ResourceContext.

144.22.8.2 public void disable() throws ResourceMonitorException

□ Disable the monitoring of this resource type for the resource context associated with this monitor instance. The resource usage is not available until it is enabled again.

Throws `ResourceMonitorException`– if the ResourceMonitor instance has been previously deleted

144.22.8.3 public void enable() throws ResourceMonitorException

□ Enable the monitoring of this resource type for the resource context associated with this monitor instance. This method SHOULD also update the current resource consumption value (to take into account all previous resource allocations and releases occurred during the time the monitor was disabled).

Throws `ResourceMonitorException`– if the ResourceMonitor instance can not be enabled (for example, some MemoryMonitor implementations evaluate the memory consumption by tracking memory allocation operation at runtime. This kind of Monitor can not get instantaneous memory value. Such Monitor instances need to be enabled at starting time.). if the ResourceMonitor instance has been previously deleted

144.22.8.4 public boolean equals(Object resourceMonitor)

resourceMonitor

□ Checks if resourceMonitor is equals to the current instance. A ResourceMonitor rm1 is equals to a ResourceMonitor rm2 if rm1.getContext().equals(rm2.getContext()) and r1.getType().equals(rm2.getType()).

Returns true if the current instance is equals to the provided resourceMonitor

144.22.8.5 public ResourceContext getContext()

□ Returns the resource context that this monitor belongs to

Returns The associated ResourceContext

144.22.8.6 public long getMonitoredPeriod()

□ Returns the time period for which the usage of this resource type is monitored.

Returns The monitored period in milliseconds, or `-1` if a monitored period is not relevant for this resource type.

144.22.8.7 public String getResourceType()

□ The name of the resource type that this monitor represents

Returns The name of the monitored resource type

144.22.8.8 public long getSamplingPeriod()

□ Returns the sampling period for this resource type.

Returns The sampling period in milliseconds, or `-1` if a sampling period is not relevant for this resource type.

144.22.8.9 public Comparable<T> getUsage() throws ResourceMonitorException

□ Returns an object representing the current usage of this resource type by this resource context.

Returns The current usage of this resource type.

Throws `ResourceMonitorException`– if the ResourceMonitor instance is not enabled.

144.22.8.10 public int hashCode()

□ Retrieves the hashCode value of this ResourceMonitor. The hashCode value is based on the hashCode value of the associated ResourceContext and the hashCode value of the type.

Returns hashcode

144.22.8.11 public boolean isDeleted()

□ Returns true if the ResourceMonitor instance has been deleted, that is the delete() method has been called previously.

Returns true if deleted.

144.22.8.12 public boolean isEnabled()

□ Checks if the monitoring for this resource type is enabled for this resource context

Returns `true` if monitoring for this resource type is enabled for this context, `false` otherwise

144.22.9 public class ResourceMonitorException extends Exception

Resource Monitor Exception reports an invalid usage of a monitor.

144.22.9.1 public ResourceMonitorException(String msg)

msg message

□ Create a new ResourceMonitorException

144.22.9.2 public ResourceMonitorException(String msg, Throwable t)

msg message

t

□ Create a new ResourceMonitorException

144.22.10 public interface ResourceMonitorFactory<T>

<T> The type for the Resource.

A Resource Monitor Factory is a service that provides Resource Monitor instances of a specific resource type (for example, CPUMonitor, MemoryMonitor...) for every Resource Context. Every Resource Monitor Factory service is registered with the RESOURCE_TYPE_PROPERTY mandatory property. This property indicates which type of Resource Monitor a Resource Monitor Factory is able to create. The type can also be retrieved through a call to getType(). The type MUST be unique (two Resource Monitor Factory instances MUST not have the same type).

144.22.10.1 public static final String RESOURCE_TYPE_PROPERTY = "org.osgi.resourcemonitoring.ResourceType"

Resource type property. The value is of type String. For example, ResourceMonitoringService.RES_TYPE_CPU

144.22.10.2 public ResourceMonitor<T> createResourceMonitor(ResourceContext resourceContext) throws ResourceMonitorException

resourceContext ResourceContext instance associated with the newly created ResourceMonitor instance

□ Creates a new ResourceMonitor instance. This instance is associated with the ResourceContext instance provided as argument (ResourceContext.addResourceMonitor(ResourceMonitor) is called by the factory). The newly ResourceMonitor instance is disabled. It can be enabled by calling ResourceMonitor.enable().

Returns a ResourceMonitor instance

Throws `ResourceMonitorException`– If the factory is unable to create a ResourceMonitor For example, when a ResourceMonitor of this type already exists for this ResourceContext

144.22.10.3 public String getType()

□ Returns the type of ResourceMonitor instance this factory is able to create.

Returns factory type

144.22.11 public interface ResourceMonitoringService

It manages the Resource Context instances. It is available through the OSGi service registry. This service holds the existing Resource Context instances. Resource Context instances are created by calling the createContext(String, ResourceContext) method.

144.22.11.1 public static final String FRAMEWORK_CONTEXT_NAME = "framework"

The name of the special, optional resource context, representing the whole OSGi framework.

144.22.11.2 public static final String RES_TYPE_CPU = "resource.type.cpu"

The name of the CPU resource type, used to monitor and control the CPU time used by a resource context. ResourceMonitoringService implementations must create CPUMonitor instances for this resource type.

144.22.11.3 public static final String RES_TYPE_DISK_STORAGE = "resource.type.disk.storage"

The name of the disk storage resource type, used to monitor and control the size of the persistent storage used by a resource context. ResourceMonitoringService implementations must create DiskStorageMonitor instances for this resource type.

144.22.11.4 public static final String RES_TYPE_MEMORY = "resource.type.memory"

The name of the memory resource type, used to monitor and control the size of the java heap used by a resource context. ResourceMonitoringService implementations must create MemoryMonitor instances for this resource type.

144.22.11.5 public static final String RES_TYPE_SOCKET = "resource.type.socket"

The name of the socket resource type, used to monitor and control the number of existing sockets used by a resource context. ResourceMonitoringService implementations must create SocketMonitor instances for this resource type.

144.22.11.6 public static final String RES_TYPE_THREADS = "resource.type.threads"

The name of the threads resource type, used to monitor and control the number of threads created by a resource context. ResourceMonitoringService implementations must create ThreadMonitor instances for this resource type.

144.22.11.7 public static final String SYSTEM_CONTEXT_NAME = "system"

The name of the Resource Context associated with System bundle (bundle 0).

144.22.11.8 public ResourceContext createContext(String name, ResourceContext template)

name The name identifying the context. Names must be unique within the framework instance.

template If a template is provided, the new resource context will inherit all resource monitoring settings (enabled monitors, thresholds) from the template.

□ Creates a new ResourceContext.

A ResourceContextEvent with type ResourceContextEvent.RESOURCE_CONTEXT_CREATED will be sent.

Returns A new ResourceContext instance.

Throws `IllegalArgumentException`– if a problem occurred, for example if the name is already used.

144.22.11.9 public ResourceContext getContext(String name)

name The resource context name

□ Returns the context with the specified resource context name.

Returns An existing ResourceContext with the specified name, or null if such a context doesn't exist

144.22.11.10 public ResourceContext getContext(long bundleId)

bundleId bundle identifier

□ Returns the ResourceContext associated to the provided bundle id.

Returns the ResourceContext associated to bundle b or null if the bundle b does not belong to a Resource Context.

144.22.11.11 public String[] getSupportedTypes()

□ Returns a list with the supported resource type names.

Returns An array containing the names of all resource types that this ResourceMonitoringService implementation supports.

144.22.11.12 public ResourceContext[] listContext()

□ Lists all available resource contexts. The list will contain the special FRAMEWORK_CONTEXT_NAME context and the SYSTEM_CONTEXT_NAME context, if it is supported.

Returns An array of ResourceContext objects, or an empty array, if no contexts have been created.

144.23 org.osgi.service.resourcemonitoring.monitor

Resource Monitoring Monitor Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.resourcemonitoring.monitor; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.resourcemonitoring.monitor; version="[1.0,1.1)"`

144.23.1 Summary

- `CPUMonitor` - A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_CPU resource type.
- `DiskStorageMonitor` - A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_DISK_STORAGE resource type.
- `MemoryMonitor` - A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_MEMORY resource type.
- `SocketMonitor` - A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_SOCKET resource type.
- `ThreadMonitor` - A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_THREADS resource type.

144.23.2 public interface CPUMonitor extends ResourceMonitor<Long>

A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_CPU resource type. CPUMonitor instance monitors the CPU consumed by a ResourceContext instance.

144.23.2.1 public long getCPUUsage()

□ Returns the CPU usage as a cumulative number of nanoseconds

The getUsage() method returns the same value, wrapped in a long.

Returns the CPU usage in nanoseconds

144.23.3 public interface DiskStorageMonitor extends ResourceMonitor<Long>

A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_DISK_STORAGE resource type. A DiskStorageMonitor instance monitors and limits the persistent storage of the bundle belonging to the ResourceContext

144.23.3.1 public long getUsedDiskStorage()

□ Returns the sum of the size of the persistent storage areas of the bundles in this resource context.

The getUsage() method returns the same value, wrapped in a long.

Returns the sum of the sizes of the persistent storage areas in bytes

144.23.4 public interface MemoryMonitor extends ResourceMonitor<Long>

A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_MEMORY resource type. A MemoryMonitor instance monitors and limits the memory used by a ResourceContext instance.

144.23.4.1 public long getMemoryUsage()

□ Returns the size of the java heap used by the bundles in this resource context.

The getUsage() method returns the same value, wrapped in a long.

Returns the size of the used java heap in bytes

144.23.5 public interface SocketMonitor extends ResourceMonitor<Long>

A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_SOCKET resource type. SocketMonitor instance are used to monitor and limit the number of in-use sockets per ResourceContext instance. SocketMonitor instance handle all types of sockets (TCP, UDP, ...).

A TCP socket is considered to be in-use when it is bound (Socket.bind(java.net.SocketAddress)) or when it is connected (Socket.connect(java.net.SocketAddress)). It leaves the in-use state when the socket is closed (Socket.close()). *

A UDP socket is in-use when it is bound (DatagramSocket.bind(java.net.SocketAddress)) or connected (DatagramSocket.connect(java.net.SocketAddress)). A UDP Socket leaves the in-use state when it is closed (DatagramSocket.close()).

144.23.5.1 public long getSocketUsage()

□ Returns the number of existing socket created by a ResourceContext.

The getUsage() method returns the same value, wrapped in a long.

Returns the number of existing socket.

144.23.6 public interface ThreadMonitor extends ResourceMonitor<Integer>

A ResourceMonitor for the ResourceMonitoringService.RES_TYPE_THREADS resource type. A ThreadMonitor instance monitors and limits the thread created by a ResourceContext instance.

144.23.6.1 public int getAliveThreads()

□ Returns the number of alive threads created by the bundles in this resource context. A Thread is considered to be alive when its java state is one of the following:

- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING

The getUsage() method returns the same value, wrapped in a int.

Returns the number of alive threads created by this resource context

144.24 References

[1] *Adaptive Monitoring of End-user OSGi based Home Boxes*
Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi. Component Base Software Engineering, 15th ACM SIGSOFT International Symposium on Component-Based Software Engineering, CBSE'2012, Bertinoro, Italy, June 2012.

[2] *Memory Monitoring in a Multi-tenant OSGi Execution Environment*
K. Attouchi, G. Thomas, A. Bottaro, and G. Muller. Proceedings of the 17th ACM SIGSOFT symposium on Component Based Software Engineering, CBSE'14, Lille, France, July 2014.

145 USB Information Device Category Specification

Version 1.0

145.1 Introduction

The *Device Access Specification* on page 39 defines a unified and sophisticated way to handle devices attached to a residential gateway or devices found in the home network by using various protocols such as USB, ZigBee, Z-Wave, KNX, UPnP, etc.

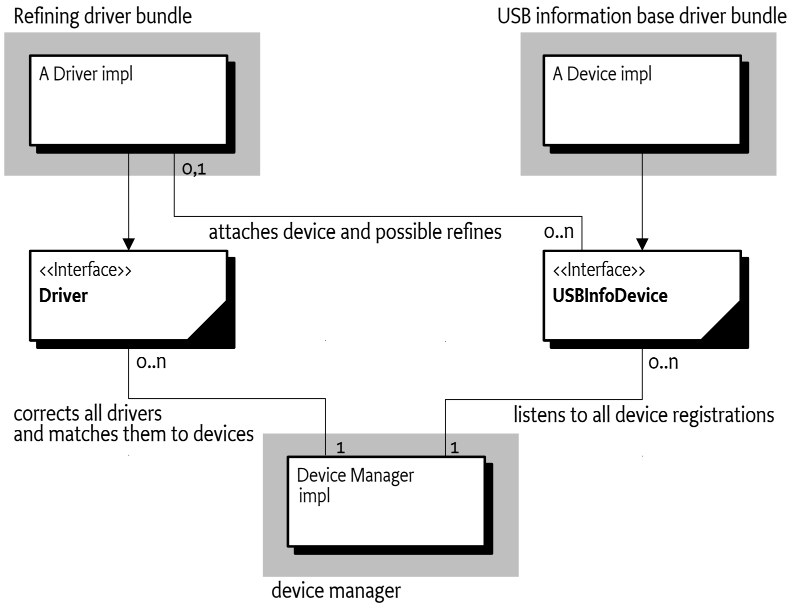
Recently, OSGi is gaining popularity as enabling technology for building embedded systems in the residential market as well as other Internet-of-Things (IoT) domains. Such systems often have USB interfaces and the need of handling USB devices attached to these systems is increasing.

Device Category Specifications on page 43 defines the concept of device categories. This specification defines a device category for USB devices.

145.1.1 Entities

- *USBInfoDevice* - The representation of a USB device. This service provide information defined by the USB Implementers Forum, Inc.

Figure 145.1 *USB Information Device Service Overview Diagram*



145.2 USBInfoDevice Service

The device services are registered in the OSGi service registry with the USBInfoDevice interface. The service is registered by a USB information base driver bundle when a USB device is attached. A USB information base driver bundle must implement USBInfoDevice interface and register the OSGi service under USBInfoDevice . Refining drivers can find USB devices via USBInfoDevice services and identify the device. The USBInfoDevice service has a set of properties.

USB Specification, see [1] *Universal Serial Bus Specification Revision 1.1* , defines that a USB device has USB interface(s). A USB information base driver bundle must register USBInfoDevice services number of USB interfaces. A USBInfoDevice service has information that contains a USB device information and a USB interface information.

The USB information base driver may need native drivers such as kernel drivers on Linux. This document has a precondition that there are native drivers. It is out of scope how to install native drivers.

145.2.1 Device Access Category

The device access category is called "USBInfo". The category name is defined as a value of DEVICE_CATEGORY constant. It must be used as a part of theDEVICE_CATEGORY service property value on the USBInfoDevice service. The category defines the following additional service properties for the USBInfoDevice service.

145.2.2 Service Properties based upon USB Specification

The USB Specification defines a Device Descriptor. USB devices report their attributes using descriptors. The following USBInfoDevice service properties use information from the USB device descriptor.

Table 145.1 Service properties of USBInfoDevice service from Device Descriptor

The key of service property	Type	Description	Device Descriptor's Field from USB Spec.
usbinfo.bcdUSB	String	OPTIONAL property key. The 4-digit BCD format. Example: "0210"	bcdUSB
usbinfo.bDeviceClass	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bDeviceClass
usbinfo.bDeviceSubClass	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bDeviceSubClass
usbinfo.bDeviceProtocol	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bDeviceProtocol
usbinfo.bMaxPacketSize0	Integer	OPTIONAL property key.	bMaxPacketSize0
usbinfo.idVendor	String	MANDATORY property key. Hexadecimal, 4-digits. Example: "0403"	idVendor

The key of service property	Type	Description	Device Descriptor's Field from USB Spec.
usbinfo.idProduct	String	MANDATORY property key. Hexadecimal, 4-digits. Example: "8372"	idProduct
usbinfo.bcdDevice	String	MANDATORY property key. The 4-digit BCD format. Example: "0200"	bcdDevice
usbinfo.Manufacturer	String	OPTIONAL property key. String value referenced by iManufacturer. The value is not the index value of iManufacturer. Example: "Buffalo Inc."	iManufacturer
usbinfo.Product	String	OPTIONAL property key. String value referenced by iProduct. The value is not the index value of iProduct. Example: "USB2.0 PC Camera"	iProduct
usbinfo.SerialNumber	String	OPTIONAL property key. String value referenced by iSerialNumber. The value is not the index value of iSerialNumber. Example: "57B0002600000001"	iSerialNumber
usbinfo.bNumConfigurations	Integer	OPTIONAL property key.	bNumConfigurations

According to the USB Specification, a device descriptor has some Interface Descriptors.

Refining drivers need each interface descriptor's bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol to identify devices. The following USBInfoDevice service properties use information from the USB interface descriptor.

Table 145.2 Service properties of USBInfoDevice service from Interface Descriptor

The key of service property	Type	Description	Interface Descriptor's Field from USB Spec.
usbinfo.bInterfaceNumber	Integer	MANDATORY property key.	bInterfaceNumber
usbinfo.bAlternateSetting	Integer	OPTIONAL property key.	bAlternateSetting
usbinfo.bNumEndpoints	Integer	OPTIONAL property key.	bNumEndpoints
usbinfo.bInterfaceClass	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bInterfaceClass
usbinfo.bInterfaceSubClass	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bInterfaceSubClass
usbinfo.bInterfaceProtocol	String	MANDATORY property key. Hexadecimal, 2-digits. Example: "ff"	bInterfaceProtocol

The key of service property	Type	Description	Interface Descriptor's Field from USB Spec.
usbinfo.Interface	String	OPTIONAL property key. String value referenced by iInterface. The value is not the index value of iInterface.	iInterface

145.2.3 Additional Service Properties

Some additional service properties are needed to identify and access a device by refining drivers.

Table 145.3 Additional service properties of USBInfoDevice service

The key of service property	Type	Description
usbinfo.bus	Integer	MANDATORY property key. The value is Integer. Used to identify USB devices with same VID / PID. The value is the ID of the USB bus assigned when connecting the USB device. USB bus ID is integer. The USB bus ID does not change while the USB device remains connected. Example: 3
usbinfo.address	Integer	MANDATORY property key. The value is Integer. Used to identify USB devices with same VID / PID. The value is the ID of the USB address assigned when connecting the USB device. USB address is integer in the range 1-127. The USB address does not change while the USB device remains connected. Example: 2

145.2.4 Match scale

When the driver service is registered by the driver bundle, the Device Manager calls match(ServiceReference) with the argument of the USBInfoDevice service's Service Reference. The driver responds with a match value based on following choices.

- MATCH_VERSION - Constant for the USB device match scale, indicating a match with USB_IDVENDOR, USB_IDPRODUCT and USB_BCDDEVICE. Value is 50.
- MATCH_MODEL - Constant for the USB device match scale, indicating a match with USB_IDVENDOR and USB_IDPRODUCT. Value is 40.
- MATCH_PROTOCOL - Constant for the USB device match scale, indicating a match with USB_BDEVICECLASS, USB_BDEVICESUBCLASS and USB_BDEVICEPROTOCOL, or a match with USB_BINTERFACECLASS, USB_BINTERFACESUBCLASS and USB_BINTERFACEPROTOCOL. Value is 30.
- MATCH_SUBCLASS - Constant for the USB device match scale, indicating a match USB_BDEVICECLASS and USB_BDEVICESUBCLASS, or a match with USB_BINTERFACECLASS and USB_BINTERFACESUBCLASS. Value is 20.
- MATCH_CLASS - Constant for the USB device match scale, indicating a match with USB_BDEVICECLASS, or a match with USB_BINTERFACECLASS. Value is 10.

145.3 Security

To acquire USB information device service, The refining bundle require that ServicePermission[USBInfoDevice, GET] is assigned.

USBInfoDevice service should only be implemented by trusted bundles. This bundle requires ServicePermission[USBInfoDevice, REGISTER].

145.4 org.osgi.service.usbinfo

USB Information Device Category Specification Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.usbinfo; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.usbinfo; version="[1.0,1.1)"
```

145.4.1 Summary

- USBInfoDevice - Represents a USB device.

145.4.2 public interface USBInfoDevice

Represents a USB device. For each USB device, an object is registered with the framework under the USBInfoDevice interface. A USB information base driver must implement this interface.

The values of the USB property names are defined by the USB Implementers Forum, Inc.

Concurrency Thread-safe

145.4.2.1 public static final String DEVICE_CATEGORY = "USBInfo"

Constant for the value of the service property DEVICE_CATEGORY used for all USB devices.

A USB information base driver bundle must set this property key.

See Also org.osgi.service.device.Constants.DEVICE_CATEGORY

145.4.2.2 public static final int MATCH_CLASS = 10

Device Access match value indicating a match with USB_BDEVICECLASS or a match with USB_BINTERFACECLASS.

145.4.2.3 public static final int MATCH_MODEL = 40

Device Access match value indicating a match with USB_IDVENDOR, and USB_IDPRODUCT.

145.4.2.4 public static final int MATCH_PROTOCOL = 30

Device Access match value indicating a match with USB_BDEVICECLASS, USB_BDEVICESUBCLASS, and USB_BDEVICEPROTOCOL or a match with USB_BINTERFACECLASS , USB_BINTERFACESUBCLASS, and USB_BINTERFACEPROTOCOL.

145.4.2.5 public static final int MATCH_SUBCLASS = 20

Device Access match value indicating a match with USB_BDEVICECLASS, and USB_BDEVICESUBCLASS or a match with USB_BINTERFACECLASS, and USB_BINTERFACESUBCLASS.

145.4.2.6 public static final int MATCH_VERSION = 50

Device Access match value indicating a match with USB_IDVENDOR, USB_IDPRODUCT, and USB_BCDDEVICE.

145.4.2.7 public static final String USB_ADDRESS = "usbinfo.address"

Service property to identify USB address.

Used to identify USB devices with same VID / PID. The value is the ID of the USB address assigned when connecting the USB device. USB address is an integer in the range 1-127 and does not change while the USB device remains connected. The value type is Integer.

145.4.2.8 public static final String USB_BALTERNATESETTING = "usbinfo.bAlternateSetting"

Service property for USB Interface Descriptor field "bAlternateSetting".

The value type is Integer. This service property is optional.

145.4.2.9 public static final String USB_BCDDEVICE = "usbinfo.bcdDevice"

Service property for USB Device Descriptor field "bcdDevice".

The value type is String; the value is in 4-digit BCD format. For example, "0200".

145.4.2.10 public static final String USB_BCDUSB = "usbinfo.bcdUSB"

Service property for USB Device Descriptor field "bcdUSB".

The value type is String; the value is in 4-digit BCD format. For example, "0210". This service property is optional.

145.4.2.11 public static final String USB_BDEVICECLASS = "usbinfo.bDeviceClass"

Service property for USB Device Descriptor field "bDeviceClass".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.12 public static final String USB_BDEVICEPROTOCOL = "usbinfo.bDeviceProtocol"

Service property for USB Device Descriptor field "bDeviceProtocol".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.13 public static final String USB_BDEVICESUBCLASS = "usbinfo.bDeviceSubClass"

Service property for USB Device Descriptor field "bDeviceSubClass".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.14 public static final String USB_BINTERFACECLASS = "usbinfo.bInterfaceClass"

Service property for USB Interface Descriptor field "bInterfaceClass".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.15 public static final String USB_BINTERFACENUMBER = "usbinfo.bInterfaceNumber"

Service property for USB Interface Descriptor field "bInterfaceNumber".

The value type is Integer.

145.4.2.16 public static final String USB_BINTERFACEPROTOCOL = "usbinfo.bInterfaceProtocol"

Service property for USB Interface Descriptor field "bInterfaceProtocol".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.17 public static final String USB_BINTERFACESUBCLASS = "usbinfo.bInterfaceSubClass"

Service property for USB Interface Descriptor field "bInterfaceSubClass".

The value type is String; the value is in 2-digit hexadecimal. For example, "ff".

145.4.2.18 public static final String USB_BMAXPACKETSIZE0 = "usbinfo.bMaxPacketSize0"

Service property for USB Device Descriptor field "bMaxPacketSize0".

The value type is Integer. This service property is optional.

145.4.2.19 public static final String USB_BNUMCONFIGURATIONS = "usbinfo.bNumConfigurations"

Service property for USB Device Descriptor field "bNumConfigurations".

The value type is Integer. This service property is optional.

145.4.2.20 public static final String USB_BNUMENDPOINTS = "usbinfo.bNumEndpoints"

Service property for USB Interface Descriptor field "bNumEndpoints".

The value type is Integer. This service property is optional.

145.4.2.21 public static final String USB_BUS = "usbinfo.bus"

Service property to identify USB bus.

Used to identify USB devices with same VID / PID. The value is the ID of the USB bus assigned when connecting the USB device. The USB bus ID is an integer and does not change while the USB device remains connected. The value type is Integer.

145.4.2.22 public static final String USB_IDPRODUCT = "usbinfo.idProduct"

Service property for USB Device Descriptor field "idProduct".

The value type is String; the value is in 4-digit hexadecimal. For example, "8372".

145.4.2.23 public static final String USB_IDVENDOR = "usbinfo.idVendor"

Service property for USB Device Descriptor field "idVendor".

The value type is String; the value is in 4-digit hexadecimal. For example, "0403".

145.4.2.24 public static final String USB_INTERFACE = "usbinfo.Interface"

Service property for name referenced by USB Interface Descriptor field "iInterface".

The value type is String. This service property is optional.

145.4.2.25 public static final String USB_MANUFACTURER = "usbinfo.Manufacturer"

Service property for name referenced by USB Device Descriptor field "iManufacturer".

The value type is String. For example, "Buffalo Inc.". This service property is optional.

145.4.2.26 public static final String USB_PRODUCT = "usbinfo.Product"

Service property for name referenced by USB Device Descriptor field "iProduct".

The value type is String. For example, "USB2.0 PC Camera". This service property is optional.

145.4.2.27 public static final String USB_SERIALNUMBER = "usbinfo.SerialNumber"

Service property for name referenced by USB Device Descriptor field "iSerialNumber".

The value type is String. For example, "57B0002600000001". This service property is optional.

145.5 References

[1] *Universal Serial Bus Specification Revision 1.1*
September 23, 1998.

146 Serial Device Service Specification

Version 1.0

146.1 Introduction

Recently, OSGi is gaining popularity as an enabling technology for building embedded systems in the residential market as well as other Internet-of-Things (IoT) domains. It is expected that communication with various devices attached to OSGi enabled gateways will be necessary.

Such communication can be implemented by means of serial connection when using non-IP devices based on ZigBee and Z-wave protocols. The most typical case arises when a USB dongle that supports such protocols is connected to the USB port of such a device, for example, residential gateway. The Operating System on the gateways will recognize the dongle as a virtual serial device and initiate a serial communication with the application process.

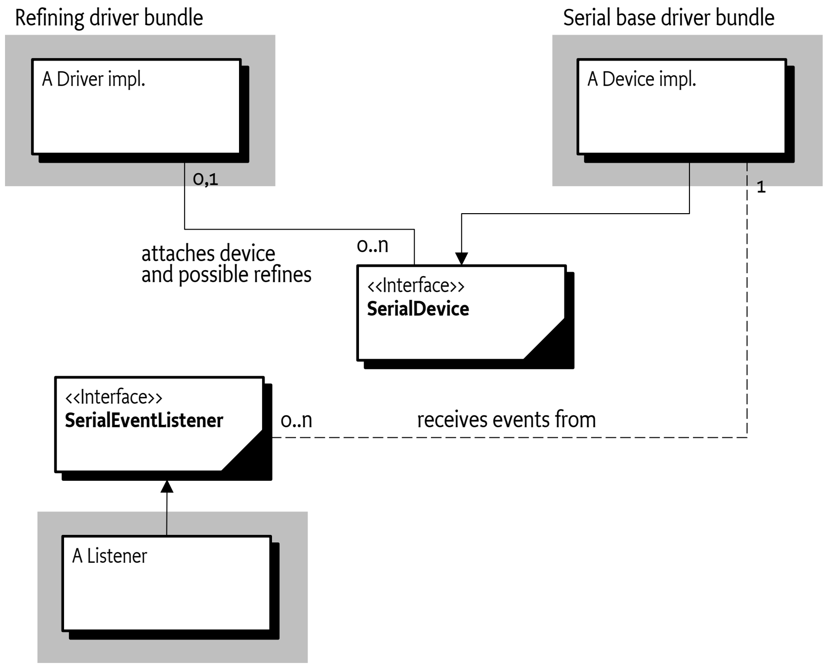
The Serial Device Service specification defines an API for establishing communications between an OSGi bundle and a serial device, such as a ZigBee coordinator or Z-Wave controller.

Device Category Specifications on page 43 defines the concept of device categories. *USB Information Device Category Specification* on page 949 defines a device category for USB devices. This specification and *USB Information Device Category Specification* on page 949 provide a solution for the USB serial use case.

146.1.1 Entities

- *SerialDevice* - This is an OSGi service that is used to represent a serial device. This OSGi service stores information regarding serial device and its status as service properties and provides communication function with the device.
- *SerialEventListener* - A listener to events coming from Serial Devices.
- *Serial base driver bundle* - The bundle that implements SerialDevice. Serial base driver bundle registers SerialDevice services with the Framework. It provides communication function with the (physical) serial devices.
- *Refining driver bundle* - Refining drivers provide a refined view of a physical device that is already represented by another Device service registered with the Framework (see the details for Device Access Specification).

Figure 146.1 Serial Device Service class diagram


Refining driver bundle
A Driver impl.
Serial base driver bundle
A Device impl.
0,1
1
attaches device and possible refines
0..n
<<Interface>> SerialDevice
<<Interface>> SerialEventListener
0..n
receives events from
A Listener

146.2 SerialDevice Service

SerialDevice is the interface expressing a serial device. It maintains information and state of the serial device as a service property. It provides the communication facility with the serial device. Each SerialDevice expresses each serial device.

SerialDevice service is registered with the service registry with service properties as shown in the following table.

Table 146.1 Service properties of SerialDevice service

The key of service property	Type	Description
DEVICE_CATEGORY	String[]	Constant for the value of the service property DEVICE_CATEGORY used for all Serial devices. Value is "Serial".
serial.comport	String	MANDATORY property key. Represents the name of the port. Examples: "/dev/ttyUSB0", "COM5", "/dev/tty.usbserial-XXXXXX"

The Serial base driver may need native libraries. This document has a precondition that there are native libraries. It is out of scope how to install native libraries.

146.3 SerialEventListener Service

Serial events are sent using the white board model, in which a bundle interested in receiving the Serial events registers an object implementing the SerialEventListener interface. A COM port name can be set to limit the events for which a bundle is notified.

146.4 USB Serial Example

The Serial base driver registers a SerialDevice service that represents a (physical) Serial device. If the device is USB Serial device, then it is recommended that the base driver implements USBInfoDevice and SerialDevice concurrently, and registers the service under USBInfoDevice and SerialDevice interfaces.

146.5 Security

To acquire the Serial device service, the refining bundle need that ServicePermission[SerialDevice, GET] are assigned.

To receive the Serial events, the bundles need that ServicePermission[SerialEventListener, REGISTER] are assigned.

SerialDevice service should only be implemented by trusted bundles. This bundle requires ServicePermission[SerialDevice, REGISTER] and ServicePermission[SerialEventListener, GET].

146.6 org.osgi.service.serial

Serial Device Service Specification Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.serial; version="[1.0,2.0)"
```

146.6.1 Summary

- SerialConstants - Constants for serial device settings.
- SerialDevice - SerialDevice is a service representing a device performing serial communication.
- SerialDeviceException - A exception used to indicate that a serial device communication problem occurred.
- SerialEvent - A serial device event.
- SerialEventListener - Serial events are sent using the white board model, in which a bundle interested in receiving the Serial events registers an object implementing the SerialEventListener interface.
- SerialPortConfiguration - An object represents the Serial port configuration.

146.6.2 public final class SerialConstants

Constants for serial device settings.

146.6.2.1 public static final int BAUD_115200 = 115200

Baud rate: 115200.

146.6.2.2 public static final int BAUD_14400 = 14400

Baud rate: 14400.

146.6.2.3 public static final int BAUD_19200 = 19200

Baud rate: 19200.

146.6.2.4 public static final int BAUD_38400 = 38400

Baud rate: 38400.

146.6.2.5 public static final int BAUD_57600 = 57600

Baud rate: 57600.

146.6.2.6 public static final int BAUD_9600 = 9600

Baud rate: 9600.

146.6.2.7 public static final int BAUD_AUTO = -1

Baud rate: Automatic baud rate (if available).

146.6.2.8 public static final int DATABITS_5 = 5

Data bits: 5.

146.6.2.9 public static final int DATABITS_6 = 6

Data bits: 6.

146.6.2.10 public static final int DATABITS_7 = 7

Data bits: 7.

146.6.2.11 public static final int DATABITS_8 = 8

Data bits: 8.

146.6.2.12 public static final int FLOWCONTROL_NONE = 0

Flow control: None.

146.6.2.13 public static final int FLOWCONTROL_RTSCTS_IN = 1

Flow control: RTS/CTS on input.

146.6.2.14 public static final int FLOWCONTROL_RTSCTS_OUT = 2

Flow control: RTS/CTS on output.

146.6.2.15 public static final int FLOWCONTROL_XONXOFF_IN = 4

Flow control: XON/XOFF on input.

146.6.2.16 public static final int FLOWCONTROL_XONXOFF_OUT = 8

Flow control: XON/XOFF on output.

146.6.2.17 public static final int PARITY_EVEN = 2

Parity: Even.

146.6.2.18 public static final int PARITY_MARK = 3

Parity: Mark.

146.6.2.19 public static final int PARITY_NONE = 0

Parity: None.

146.6.2.20 public static final int PARITY_ODD = 1

Parity: Odd.

146.6.2.21 public static final int PARITY_SPACE = 4

Parity: Space.

146.6.2.22 public static final int STOPBITS_1 = 1

Stop bits: 1.

146.6.2.23 public static final int STOPBITS_1_5 = 3

Stop bits: 1.5.

146.6.2.24 public static final int STOPBITS_2 = 2

Stop bits: 2.

146.6.3 public interface SerialDevice

SerialDevice is a service representing a device performing serial communication.

Concurrency Thread-safe

146.6.3.1 public static final String DEVICE_CATEGORY = "Serial"

Constant for the value of the service property DEVICE_CATEGORY used for all Serial devices.

A Serial base driver bundle must set this property key.

See Also org.osgi.service.device.Constants.DEVICE_CATEGORY

146.6.3.2 public static final String SERIAL_COMPORT = "serial.comport"

Service property for the serial comport.

Represents the name of the port. The value type is String.

For example, "/dev/ttyUSB0", "COM5", or "/dev/tty.usbserial-XXXXXX".

146.6.3.3 public SerialPortConfiguration getConfiguration()

□ Gets the Serial port configuration.

Returns The SerialPortConfiguration object containing the configuration.

146.6.3.4 public InputStream getInputStream() throws IOException

□ Returns an input stream.

Returns An input stream.

Throws IOException– if an I/O error occurred.

146.6.3.5 public OutputStream getOutputStream() throws IOException

□ Returns an output stream.

Returns An output stream.

Throws IOException– If an I/O error occurred.

146.6.3.6 public boolean isCTS()

□ Returns the CTS state.

Returns The CTS state.

146.6.3.7 public boolean isDSR()

□ Returns the DSR state.

Returns The DSR state.

146.6.3.8 public boolean isDTR()

□ Returns the DTR state.

Returns The DTR state.

146.6.3.9 public boolean isRTS()

□ Returns the DTS state.

Returns The DTS state.

146.6.3.10 public void setConfiguration(SerialPortConfiguration configuration) throws SerialDeviceException

configuration The SerialPortConfiguration object containing the configuration.

□ Sets the Serial port configuration.

Throws SerialDeviceException– If the parameter is specified incorrectly or the parameter is not supported.

146.6.3.11 public void setDTR(boolean dtr) throws SerialDeviceException

dtr true for DTR on; false for DTR for off.

□ Sets the DTR state.

Throws SerialDeviceException– If the parameter is not supported.

146.6.3.12 public void setRTS(boolean rts) throws SerialDeviceException

rts true for RTS on; false for RTS for off.

□ Sets the RTS state.

Throws SerialDeviceException– If the parameter is not supported.

146.6.4 public class SerialDeviceException extends Exception

A exception used to indicate that a serial device communication problem occurred.

146.6.4.1 public static final int PORT_IN_USE = 1

The port in use.

146.6.4.2 public static final int UNKNOWN = 0

The reason is unknown.

146.6.4.3 public static final int UNSUPPORTED_OPERATION = 2

The operation is unsupported.

146.6.4.4 public SerialDeviceException(int type, String message)

type The type for this exception.

message The message.

□ Creates a SerialDeviceException with the specified type and message.

146.6.4.5 public int getType()

□ Returns the type for this exception.

Returns The type of this exception.

146.6.5 public interface SerialEvent

A serial device event. SerialEvent objects are delivered to SerialEventListeners when an event occurs.

A type of code is used to identify the event. Additional event types may be defined in the future.

Concurrency Thread-safe

146.6.5.1 public static final int DATA_AVAILABLE = 1

Event type indicating data available.

146.6.5.2 public String getComPort()

□ Returns the port name of this event.

This value must be equal to the value of SerialDevice.SERIAL_COMPORT service property of the SerialDevice.

Returns The port name of this event.

146.6.5.3 public int getType()

□ Returns the type of this event.

Returns The type of this event.

146.6.6 public interface SerialEventListener

Serial events are sent using the white board model, in which a bundle interested in receiving the Serial events registers an object implementing the SerialEventListener interface. A COM port name can be set to limit the events for which a bundle is notified.

Concurrency Thread-safe

146.6.6.1 public static final String SERIAL_COMPORT = "serial.comport"

Key for a service property that is used to limit received events.

146.6.6.2 public void notifyEvent(SerialEvent event)

event The SerialEvent object.

□ Callback method that is invoked for received an event.

146.6.7 public class SerialPortConfiguration

An object represents the Serial port configuration.

Concurrency Immutable

146.6.7.1 public SerialPortConfiguration(int baudRate, int dataBits, int flowControl, int parity, int stopBits)

baudRate Baud rate.

dataBits Data bits.

flowControl Flow control.

parity Parity.

stopBits Stop bits.

□ Creates an instance of the serial port configuration with the specified Baud rate, Data bits, Flow control, Parity and Stop bits.

146.6.7.2 public SerialPortConfiguration(int baudRate)

baudRate Baud rate.

□ Creates an instance of the serial port configuration with the specified Baud rate and the following configuration: Data bits = 8, Flow control = none, Parity = none, Stop bits = 1.

146.6.7.3 public SerialPortConfiguration()

□ Creates an instance of the serial port configuration with the following configuration: Baud rate = auto, Data bits = 8, Flow control = none, Parity = none, Stop bits = 1.

146.6.7.4 public int getBaudRate()

□ Returns the baud rate.

Returns The baud rate.

146.6.7.5 public int getDataBits()

□ Returns the data bits.

Returns The data bits.

146.6.7.6 public int getFlowControl()

□ Returns the flow control.

Returns The flow control.

146.6.7.7 public int getParity()

□ Returns the parity.

Returns The parity.

146.6.7.8 public int getStopBits()

□ Returns the stop bits.

Returns The stop bits.

147 Transaction Control Service Specification

Version 1.0

147.1 Introduction

Software Transactions are an important aspect of most modern applications. The job of a Transaction is to ensure logical consistency for units of work within the application. Any time that the application accesses a persistent external resource then a Transaction ensures that the set of changes made to the resource(s) are Atomic, Consistent, Isolated, and Durable (ACID).

There are a variety of techniques for managing the lifecycle of software Transactions used in an application. The most primitive mechanisms are for the application code to directly interact with the Transaction Manager, but higher level abstractions can automatically manage the lifecycle of Transactions through the use of Aspect Oriented Programming. Whatever techniques are used to manage the Transaction lifecycle it is also necessary for any resource access that occurs within the Transaction to be registered with the Transaction manager. As with managing the Transaction lifecycle, this work may be performed by the client, or by a an intermediate framework without direct action from the client.

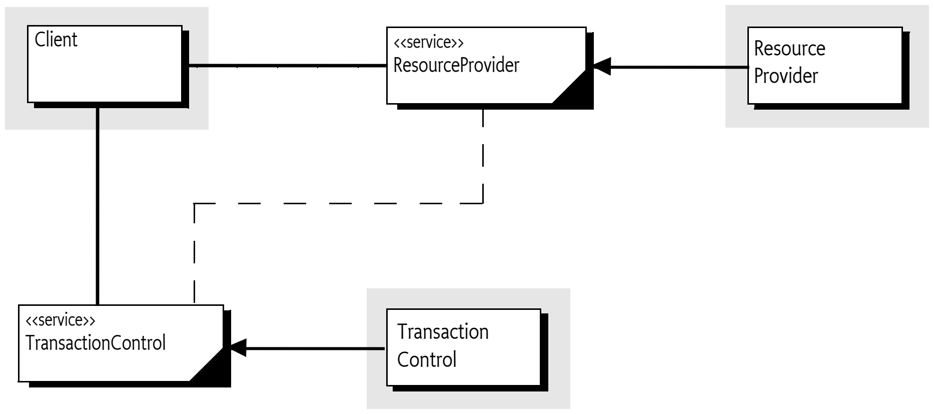
OSGi applications consist of a set of independent modules which interact via the OSGi service registry; as such there is no single container which can be relied upon to manage the range of tasks needed to successfully use a Transaction. This leaves OSGi clients with little choice but to depend on specific environments, sacrificing portability, or to directly use Transactions via the *JTA Transaction Services Specification* on page 541. The purpose of the Transaction Control Service is twofold:

- To enable a portable, modular abstraction for Transaction lifecycle management
- To allow different resource types to be easily used within a Transaction

147.1.1 Essentials

- *Scoped Work* - A function or code block with an associated execution context, known as a Scope. The Scope may be *Transactional*, that is, associated with a Transaction, or a *No Transaction Scope*, that is, with no associated Transaction.
- *Client* - Application code that wishes to invoke one or more pieces of Scoped Work.
- *Transaction Control Service* - The OSGi service representing the Transaction Control Service implementation. Used by the Client to execute pieces of Scoped Work.
- *Resource* - A local or remote software component which is stateful and can participate in a transaction.
- *Resource Provider* - A service or object which provides managed access to a Scoped Resource, that is, a managed connection to the Resource which integrates with ongoing Transactions as necessary.
- *Transaction Context* - A Java object representing the state of a Scope

Figure 147.1 *Class and Service overview*



147.1.2 Entities

- *Transaction Control Service* - A service that can execute pieces of work within a Scope, and be queried to establish the current Scope.
- *Client* - The code that requests for Work to be run in a particular Scope.
- *Work* - A collection of instructions that interact with zero or more Resources within a Scope
- *Scoped Resource* - A resource connection with a managed lifecycle. The connection will automatically participate in Transactions associated with Transactional Scopes, and its lifecycle is tied to the Scope within which it is used.

147.2 Usage

This section is an introduction in the usage of the Transaction Control Service. It is not the formal specification, the normative part starts at *Transaction Control Service* on page 969. This section leaves out some of the details for clarity.

147.2.1 Synopsis

The Transaction Control Service provides a mechanism for a client to run work within a defined Scope. Typically a Scope is also associated with a Transaction. The purpose of a Scope is to simplify the lifecycle of resources, and to allow those resources to participate in any ongoing Transaction. Any Scoped Resources accessed during a Scope will remain available throughout the scope, and be automatically cleaned up when the Scope completes.

Each Scope is started by the Client by passing piece of work to the Transaction Control Service. The transaction control service will then begin a scope if needed, execute the work, and then complete the scope if needed. The different methods on the Transaction Control Service provide different lifecycle semantics for the Scope. Some methods establish a Transactional Scope, others may suspend an active Transactional Scope replacing it with a No Transaction Scope.

When a piece of Scoped Work is executing it may access one or more Scoped Resources. When a Scoped Resource is first accessed within a Scope it is bound to that Scope so that future accesses use the same physical resource. At the end of the Scope the resource is detached from the scope and the physical resource is released. If the Scope is Transactional then the Scoped Resource will also participate in the transaction.

At the end of a piece of Scoped Work the Scope is finished. For a No Transaction Scope this simply involves calling any registered callbacks. For a Transactional Scope, however, the Transaction must be completed or rolled back. If the Scoped Work exits normally, and no call has been made to force the Transaction to roll back, then the Transaction will commit. If, however, the Work exits with an Exception or the Transaction has been marked for roll back, then the Transaction will roll back. The result of the Work then flows back to the caller in an appropriate way.

147.2.2 Running Scoped Work

The general pattern for a client is to obtain the Transaction Control Service and one or more Resource Provider instances. The Resource Provider(s) may come from the Service Registry, or from a Factory, and are used to create Scoped Resource instances. These instances can then be used in the scoped work. This is demonstrated in the following example:

```
@Reference
TransactionControl control;

Connection connection;

@Reference
void setResourceProvider(JDBCConnectionProvider provider) {
    connection = provider.getResrouce(control)
}

public void addMessage(String message) {
    control.required(() -> {
            PreparedStatement ps = connection.prepareStatement(
                            "Insert into TEST_TABLE values ( ? )");
                    ps.setString(1, message);
                    return ps.executeUpdate();
        });
}

public List<String> listMessages(String message) {
    control.notSupported(() -> {
            List<String> results = new ArrayList<String>();
            ResultSet rs = connection.createStatement()
                    .executeQuery("Select * from TEST_TABLE");
            while(rs.next()) {
                results.add(rs.getString(1));
            }
            return results;
        });
}
```

This example demonstrates how simply clients can execute scoped work using the Transaction Control Service. In this case write operations always occur in a Transactional Scope, but read operations may occur in a Transactional Scope *or* a No Transaction Scope. In all cases the lifecycle of the underlying connection is automatically managed, and there is no need to close or commit the connection.

147.2.3 Accessing Scoped Resources

The Transaction Control Service can be used to manage the Scope of any piece of Work, but Scopes are primarily used to simplify resource lifecycle management when using Scoped Resources. A Scoped Resource is created using a Resource Provider, and the returned object can then be used in any scope to access the associated Resource.

The example in *Running Scoped Work* on page 967 uses a `JDBCConnectionProvider`, which is a specialization of the generic `ResourceProvider` interface that returns JDBC `Connection` objects. Other specializations of the Resource Provider exist in this specification, and third party providers may provide their own specializations for proprietary resource types.

Once a Resource Provider has been obtained, a Scoped Resource is created from it by passing the Transaction Control Service to the getResource method. This returns the Scoped Resource object that can then be used in Scoped Work.

147.2.4 Exception Management

One of the most significant sources of error in applications that use transactions is caused by incorrect Exception Handling. These errors are the primary reason for using a framework or container to manage transactions, rather than trying to manage them in the application code.

Exceptions tend to be more common in code that makes use of transactions because the code is usually performing actions that may fail, for example making updates to a database. Also, many of these exceptions (such as java.sql.SQLException) are checked exceptions. As Scoped Work will typically raise both checked and unchecked exceptions it is defined as a Callable. As the callable interface throws Exception it is not necessary to catch or wrap any exception generated within Scoped Work.

```
// An SQLException may be raised by the query,
// but we don't need to catch it
control.required(() -> connection.createStatement()
    .executeQuery("Insert into TEST_TABLE values ( 'Hello World!' )"));
```

An exception indicates that a problem has occurred in a piece of code therefore, by default, any exception thrown from inside a Transactional Scope will cause the Transaction to roll back. This means that the Scoped Work can safely ignore any updates that were made in the event of an exception.

147.2.4.1 Handling Exceptions

Scoped Work is free to throw checked or unchecked exceptions, however these exceptions cannot be directly thrown on by the Transaction Control Service. The primary reason for this is that directly rethrowing the exception would force users of the Transaction Control Service to either:

- Declare throws Exception on the calling method
- Add try/catch Exception blocks around the calls to the Transaction Control Service.

Both of these solutions are undesirable, as they force unnecessary boilerplate code, and potentially shadow real checked exceptions in the API. Exceptions generated as part of Scoped Work are therefore wrapped by the Transaction Control Service in a ScopedWorkException. ScopedWorkException is an unchecked exception and so can be ignored if no special handling is required.

In the case where the callers API requires the unwrapped exception type to be thrown a ScopedWorkException can be easily unwrapped using the as method.

```
try {
    control.required(() -> connection.createStatement()
        .executeQuery("Insert into TEST_TABLE values ( 'Hello World!' )"));
} catch (ScopedWorkException swe) {
    // This line throws the cause of the ScopedWorkException as
    // an SQLException or as a RuntimeException if appropriate
    throw swe.as(SQLException.class);
}
```

If there is more than one potential checked Exception type that should be rethrown then the asOneOf method can be used.

```
try {
    control.required(() -> connection.createStatement()
        .executeQuery("Insert into TEST_TABLE values ( 'Hello World!' )"));
} catch (ScopedWorkException swe) {
    // This line throws the cause of the ScopedWorkException as
    // an SQLException or as a RuntimeException if appropriate
    throw swe.asOneOf(SQLRecoverableException.class, SQLTransientException.class);
}
```

147.2.4.2 Avoiding Transaction Rollback

In general if a piece of Work running in a Transactional Scope exits with an exception the associated Transaction will roll back. Sometimes, however, certain exception types should not cause the Transaction to roll back. This can be indicated to the Transaction Control Service when the Scope is being declared.

```
control.build()
    .noRollbackFor(URISyntaxException.class)
    .required(() -> {
            ...
        });
```

In this example the Transaction does not roll back for any `URISyntaxException`. Sometimes this is too coarse grained, and the Transaction should only avoid rolling back for one specific exception instance. In this case the instance can be passed to the Transaction Control Service `ignoreException` method.

```
control.required(() -> {
        try {
            // A URISynaxException from here is safe
            ...
        } catch (URISyntaxException e) {
            control.ignoreException(e);
            throw e;
        }
        // A URISynaxException from here is *not* safe
        ...
    });
```

147.2.5 Multi Threading

By its very definition a Scope is associated with a single piece of Work, and therefore a single thread. If a piece of Scoped Work starts new threads, or submits tasks to other threads, then any code executed on those threads will not occur within the Scope.

Scoped Resources are always thread-safe, and can be used concurrently in different Scopes. This is true even if the underlying physical resources are not thread safe. It is the responsibility of the Scoped Resource implementation to ensure that the underlying physical resources are protected correctly.

147.3 Transaction Control Service

The Transaction Control Service is the primary interaction point between a client and the Transaction Control Service implementation. A Transaction Control Service implementation must expose a service implementing the `TransactionControl` interface.

Clients obtain an instance of the Transaction Control Service using the normal OSGi service registry mechanisms, either directly using the OSGi framework API, or using dependency injection.

The Transaction Control Service is used to:

- Execute work within a defined scope
- Query the current execution scope
- Associate objects with the current execution scope
- Register for callbacks when the scope ends
- Enlist resource with the current transaction (if there is a Transaction Scope active)
- Mark the current scope for rollback (if there is a Transaction scope)

147.3.1 Scope Life Cycle

The life cycle of a scope is tied to the execution of a piece of scoped work. Unless a scope is being inherited then a scope starts immediately before the scoped work executes and ends immediately after the scoped work completes, even if the scoped work throws an exception.

The first action that a client wishing to execute scoped work must take is to identify the type of scope that they wish to use. The work should then be passed to the relevant method on the TransactionControl service:

Table 147.1 Methods for executing scoped work

Method Name	Existing Scope	Description
required(Callable)	Unscoped	Begins a new Transaction scope and executes the work inside it
required(Callable)	No Transaction scope	Suspends the No Transaction Scope and begins a new Transaction scope, executing the work inside it. After the work completes the original scope is restored.
required(Callable)	Transaction scope	Runs the work within the existing scope
requiresNew(Callable)	Unscoped	Begins a new Transaction scope and executes the work inside it
requiresNew(Callable)	No Transaction scope	Suspends the No Transaction Scope and begins a new Transaction scope, executing the work inside it. After the work completes the original scope is restored.
requiresNew(Callable)	Transaction scope	Suspends the Transaction Scope and begins a new Transaction scope, executing the work inside it. After the work completes the original scope is restored.
supports(Callable)	Unscoped	Begins a new No Transaction scope and executes the work inside it
supports(Callable)	No Transaction scope	Runs the work within the existing scope
supports(Callable)	Transaction scope	Runs the work within the existing scope
notSupported(Callable)	Unscoped	Begins a new No Transaction scope and executes the work inside it
notSupported(Callable)	No Transaction scope	Runs the work within the existing scope
notSupported(Callable)	Transaction scope	Suspends the Transaction Scope and begins a new No Transaction scope, executing the work inside it. After the work completes the original transaction scope is restored.

Once the relevant method has been identified the client passes the scoped work to the Transaction Control Service. In the typical case the Transaction Control Service must then:

1. Establish a new scope
2. Execute the scoped work
3. Finish the scope, calling any registered callbacks and committing the Transaction if the scope is a Transaction Scope
4. Return the result of the scoped work to the client

The Transaction Control Service must only finish a scope once, after the execution of the Scoped Work which originally started the scope. This means that callbacks registered by a piece of Scoped Work may not run immediately after the work finishes, but will be delayed until the parent task has finished if the scope was inherited.

147.3.2 Scopes and Exception Management

Resource access is intrinsically error-prone, and therefore there are many potential failure scenarios. Exceptions therefore form an important part of the scope lifecycle.

147.3.2.1 Client Exceptions

The work provided by the client to the Transaction Control Service is passed as a Callable, meaning that the work may throw an Exception. An Exception thrown by the work is known as a *Client Exception.*

If a client exception is thrown then it must be caught by the Transaction Control Service and handled appropriately by finishing the scope as required. Once the scope has completed the client exception must be wrapped in a ScopedWorkException and rethrown by the Transaction Control service.

If a number of scopes are nested then a ScopedWorkException may be received as a client Exception. A ScopedWorkException must not be re-wrapped by the Transaction Control Service using the normal Exception chaining mechanism, but instead a new ScopedWorkException must be created initialized with the original cause. The caught ScopedWorkException must then be added to the new ScopedWorkException as a suppressed Exception. This prevents clients from having to deeply introspect the exception cause chain to locate the original error.

147.3.2.2 Rethrowing Client Exceptions

In the general case clients will not need to catch a ScopedWorkException, and it can be left to report/handle at a higher level. Sometimes, however, the Exceptions thrown by a piece of work represent an important part of the API, and they need to be thrown on without being wrapped in a ScopedWorkException. The ScopedWorkException provides a simple mechanism to do this. The client simply calls one of the asOneOf(Class,Class) methods which will throw the cause of the Exception as one of the supplied checked Exception types, or directly as an unchecked Exception if the cause is unchecked.

The asOneOf() methods always throw an Exception, but the method return value is declared as a RuntimeException. This can be used to simplify the act of rethrowing the cause when using this method.

```
try {
    txControl.required(() -> {
                // Do some work in here that may throw IOException
                // or ClassNotFoundException
                return result;
        });
} catch (ScopedWorkException swe) {
    throw swe.asOneOf(IOException.class, ClassNotFoundException.class);
}
```

If the cause of a ScopedWorkException is a checked exception, but that exception is not assignable to any of the types passed to the asOneOf() method then the cause of the ScopedWorkException will still be thrown, however there will be no compiler assistance for the user when writing their throws clause.

147.3.2.3 Exceptions Generated by the Transaction Control Service

Many operations performed by the Transaction Control Service, particularly when finishing a scope, may result in an Exception. Internal failures, for example a failure when attempting to commit a resource, must be wrapped in a TransactionException and thrown to the client.

A TransactionException must never override a ScopedWorkException. In the case where a ScopedWorkException should be thrown and a Transaction Control Service failure occurs then the TransactionException must be set as a suppressed exception in the ScopedWorkException.

147.3.3 Transaction Scope lifecycle

In addition to callbacks and scoped variables Transaction scopes also provide an ongoing software transaction which shares the lifecycle of the scope. There are therefore additional lifecycle rules for Transaction Scopes

147.3.3.1 Triggering Rollback in Transaction Scopes

By default a transaction will commit automatically when the piece of work completes normally. If this is not desired (for example if the work's business logic determines that the transaction should not complete) then the work may trigger a rollback in one of two ways.

Calling setRollbackOnly() on the Transaction Control object will mark the transaction for rollback so that it will never commit, even if the method completes normally. This is a one-way operation, and the rollback state can be queried using getRollbackOnly()

```
txControl.required(() -> {
            // Do some work in here
            ...
            // This work will not be committed!
            txControl.setRollbackOnly();
            return result;
    });
```

Throwing an exception from the piece of work will, by default, cause the transaction to be rolled back. Note that this is different from Java EE behavior, where a checked exceptions *does not* trigger rollback. This is a deliberate difference as many applications get the wrong behavior based on this default. For example SQLException is a commonly thrown Exception in JDBC, but is rarely, if ever, a “safe return”. Forgetting to override this behavior means that production code will fail to enforce the correct transaction boundaries.

```
txControl.required(() -> {
            // Do some work in here
            ...
            // Uh oh - something went wrong!
            throw new IllegalStateException(“Kaboom!”);
    });
```

147.3.3.2 Avoiding Rollback

Sometimes it is preferable for a piece of work to throw an exception, but for that exception not to trigger a rollback of the transaction. For example some business exceptions may be considered “normal”, or it may be the case that the work performed so far must be persisted for audit reasons.

There are two ways to prevent a transaction from rolling back when a particular exception occurs

The Transaction Control service provides a TransactionBuilder. The builder can be used to define sets of Exception types that should, or should not, trigger rollback. The most specific match will be used to determine whether the transaction should roll back or not.

The Transaction Control service provides an ignoreException(Throwable) method. This can be used from within an Active Transaction to declare a specific Exception object that should not trigger rollback.

If a transaction is marked for rollback using setRollbackOnly() then it must roll back, even if the work throws an exception which would not normally trigger a rollback.

147.3.3.3 Rollback in inherited transactions

If a piece of scoped work inherits a transaction scope then the transaction is not committed until the inherited scope completes. Therefore if the nested scoped work throws an exception then this must mark the transaction for rollback, unless the exception has been explicitly ignored or configured not to cause rollback.

Any exception thrown by the nested scoped work must result in a ScopedWorkException in exactly the same way as it would when not nested.

```
txControl.required(() -> {
            // Do some work in here
            ...
            try {
                txControl.required(() -> {
                            // The outer transaction is inherited
                            throw new RuntimeException();
                    });
            } catch (ScopedWorkException swe) {
                // The transaction is still active, but now marked for rollback
            }
    });
```

147.3.3.4 Read Only transactions

Resources accessed within a transaction are frequently used to update persistent data, however in some cases it is known in advance that no changes will be made to the data. In the case where no changes are going to be made then different, more optimal, algorithms can be used by the resource to improve performance. It is therefore useful for applications to be able to indicate when resources are going to be used in a read-only way.

To indicate that a transaction is read-only the TransactionBuilder must be used.

```
txControl.build()
    .readOnly()
    .required(() -> {
            // Do some work in here
            ...
            return result;
        });
```

The readOnly method provides a hint to the TransactionControl service that the scoped work only uses read access to resources. The TransactionControl service is free to ignore this hint if it does not offer read-only optimizations. Also, read-only only applies to Transaction Scopes. No Transaction Scopes always ignore the call to readOnly.

147.3.3.4.1 Determining whether a Transaction is read only

The TransactionContext provides access to whether the transaction is read only using the isReadOnly() method. This method will return true if the transaction was started using the read only flag, and the TransactionControl service supports read-only optimization.

This method is primarily available so that resource providers can set their read-only status correctly when they first enlist with the transaction. Resource providers are free to ignore the read only status as it is provided for optimization only.

147.3.3.4.2 Writing to resources using in a read only transaction

When a client begins a transaction in read-only mode there is no API restriction that prevents them from writing to one or more resources. If the scoped work does write to the resource then the result is undefined. The write may succeed, or it may result in an exception, triggering a rollback.

Clients should avoid declaring a transaction as read only unless they are certain that no resources are updated within the scope of the work. This includes any operations performed by external services which inherit the transaction.

147.3.3.4.3 Changing the read state in nested transactions

When a client begins a Transaction Scope using the required method then it inherits the existing Transaction Scope if it exists. It is not possible to change the writability of an inherited transaction.

In the case where the inherited transaction is a writable transaction then the `readOnly()` state declared for the nested scope will be ignored. In the case where the inherited transaction is read only then an attempt to change the transaction to a writable transaction will fail with a TransactionException.

If the nested transaction is declared using requiresNew then it will create a new transaction which may have a different writability from the outer scope.

147.4 The TransactionContext

When a client uses the TransactionControl service to scope a piece of work, the scope gains an associated Transaction Context. The current transaction context is not normally needed by clients, but is an important integration point for ResourceProviders, and for clients that wish to register transaction completion callbacks.

The Transaction Control Service provides methods that can be used to query the current transaction context.

- `activeTransaction()` - returns true if there is a Transaction scope associated with the currently executing work.
- `activeScope()` - returns true if there is a Transaction Scope or a No Transaction Scope associated with the currently executing work.
- `getCurrentContext()` - returns the current `TransactionContext`, or `null` if the currently executing code is unscoped. If the current work has a No Transaction scope then the returned Transaction Context will report its status as `NO_TRANSACTION`

If a Transaction scope is active then it may either be backed by a Local Transaction, or by an XA Transaction, which affects the types of resource that can be used with the Transaction Context. The transaction support can be queried using the `supportsLocal()` and `supportsXA()` methods on the transaction context object. Some implementations may support both XA and Local resources in the same transaction, but these are still considered to be XA Transactions.

147.4.1 Transaction Lifecycle callbacks

In addition to registering Resources with the Transaction Context clients or resources may register callback functions. Callback functions can run either before or after the transaction finishes, depending as to whether they are registered using `preCompletion(Runnable)` or `postCompletion(Consumer)` to register their callbacks.

Lifecycle callbacks may be registered at any point during the execution of scoped work. Once the scoped work has finished it is no longer possible to register a pre-completion callback (for example inside another lifecycle callback). Attempts to register a pre-completion callback outside the execution of the scoped work must fail with an IllegalStateException. Post-completion callbacks may be also be registered with the Transaction Context after the scoped work completes, up to the point where the first post-completion callback is called. Specifically a pre-completion callback, or a resource participating in the transaction may register a post-completion callback. Attempts to register a post-completion callback after this must fail with an IllegalStateException.

147.4.1.1 Pre-completion Callbacks

Pre-completion callbacks run immediately after the end of the scoped work, and before any associated transaction finishes. Because pre-completion callbacks run before the end of the transaction they are able to prevent it from committing, either by calling setRollbackOnly() or potentially by throwing a RuntimeException. If the scope is a No Transaction scope then there is no commit to prevent.

If scoped work completes with an exception that triggers rollback, then the Transaction Context must be marked for rollback before calling any pre-completion callbacks.

Exceptions generated by pre-completion callbacks are gathered, If any of the generated Exceptions would trigger rollback then the transaction is treated as having failed with the first of those exceptions. Any other exceptions are added as suppressed exceptions. Assuming that no Client Exception occurred then the failure must be reported by throwing a TransactionRolledBackException, or in the case of a No Transaction scope, a TransactionException.

147.4.1.2 Post-completion Callbacks

Post-completion callbacks are run after any associated transaction finishes. As the transaction has completed, post-completion callbacks receive the completion state of the transaction as a method parameter. In the case of a No Transaction context there is no transaction, so the post-completion callbacks immediately follow the pre-completion callbacks, and are passed a status of NO_TRANSACTION.

Exceptions generated by post-completion callbacks are unable to affect the outcome of any transaction, and must therefore be logged, but not acted on further by the Transaction Control service.

Although Post-completion callbacks run after the transaction, the Transaction Context must still be valid when they execute. In particular post-completion callbacks must have access to any scoped variables registered with the Transaction Context

147.4.2 Scoped variables

A Transaction context may be used to store scoped variables. These variables are attached to the TransactionContext, and will be released after the Context finishes. Scoped resources are guaranteed to be accessible in lifecycle callbacks.

Variables may be added to the scope using putScopedValue(Object,Object) and retrieved using getScopedValue(Object). These methods are valid both for Active Transactions and the No Transaction scope.

147.4.3 Transaction Key

Every Active Transaction has an associated key, which will be unique within the lifetime of the TransactionControl service's registration. That is, a registered Transaction Control instance will never reuse a key. The key object is opaque, but is guaranteed to be suitable for use as a key in a HashMap. Note that the Transaction Key is not globally unique, but only unique to the registered TransactionControl service. In particular, two concurrently registered TransactionControl services may simultaneously use the same key, and/or a Transaction Control implementation may reuse keys if it unregisters and then re-registers its service with a different service id.

TransactionContexts for the No Transaction scope have a null key.

147.4.4 The Transaction Status

The current state of a Transaction Context is represented by a Java enum, and can be queried by calling getTransactionStatus(). The status of a Transaction Context will change over time until it reaches a terminal state. Once a terminal state has been reached the status of the Transaction Context will not change again.

The status of a Transaction Context will always move forward through the enum values, that is, the status can never move from one state to another state with a lower sort order. Note that a Transaction Context will not necessarily enter all of the intermediate states between two values.

Table 147.2 Transaction Status Values

Status	Terminal	Description
NO_TRANSACTION	yes	This Transaction Context is for a No Transaction Scope
ACTIVE	no	This Transaction Scope is executing and not marked for rollback
MARKED_ROLLBACK	no	This Transaction Scope is executing and has been marked for rollback
PREPARING	no	A two phase commit is occurring and the transaction is being prepared. This state is visible during the prepare calls on XA resources
PREPARED	no	A two phase commit is occurring and the transaction has been prepared. This state is visible immediately prior to committing or rolling back XA resources
COMMITTING	no	The transaction is being committed. This state is visible during the commit calls on resources
COMMITTED	yes	The transaction was successfully committed.
ROLLING_BACK	no	The transaction is being rolled back. This state is visible during the rollback calls on resources
ROLLED_BACK	yes	The transaction was successfully rolled back.

147.4.5 Local Transaction scopes

A Local Transaction is not persistent, and therefore not recoverable. It also may not be atomic or consistent if multiple resources are involved. Local transactions do, however, provide isolation and durability, even when multiple resources are involved.

A Local Transaction is therefore a very good choice when a single resource is involved as it is extremely lightweight and provides ACID behavior. Local Transactions do provide benefits when multiple resources are involved, however it is important to realize that Local Transactions may end up in a state where some commits have succeeded and others failed.

147.4.5.1 The Local Transaction Lifecycle

The transaction context for a local transaction begins in the ACTIVE state, and may enter the MARKED_ROLLBACK state if the client calls setRollbackOnly().

A local transaction must always return true from the supportsLocal() method, indicating that LocalResource participants may be registered using the registerLocalResource(LocalResource) method.

Once the transactional work has completed and the pre-completion callbacks have run the transaction will be proceed as follows:

Table 147.3 Lifecycle rules for Local Transactions

Active	Marked for Rollback
1. Set the Transaction Status to COMMITTING 2. Call commit on the first LocalResource 3. If the first commit fails set the status Transaction Status to ROLLING_BACK and initialize a TransactionRolledBackException with its cause set to the failure. 4. Continue committing or rolling-back resources based on the Transaction Status. If a failure occurs then add it as a suppressed exception of an existing TransactionException, creating a new TransactionException if this is the first failure. 5. Set the Transaction Status to COMMITTED or ROLLED_BACK as appropriate 6. Call the post-completion callbacks, passing the Transaction Status	1. Set the Transaction Status to ROLLING_BACK 2. Call rollback on each of the LocalResources 3. If a failure occurs then add it as a suppressed exception of an existing TransactionException, creating a new TransactionException if this is the first failure. 4. Set the Transaction Status to ROLLED_BACK 5. Call the post-completion callbacks, passing the Transaction Status

147.4.5.2 Local Transaction Support Service Properties

A TransactionControl Service which supports local transactions may be identified using the `osgi.local.enabled` property which will be set to `Boolean.TRUE`.

147.4.6 XA Transaction scopes

An XA transaction is persistent, and therefore can be recoverable. It is also atomic and consistent even if multiple resources are involved.

An XA Transaction is therefore a very good choice when a multiple resource are involved as it provides ACID behavior. XA transactions are, however, more heavyweight than local transactions, and should only be used where they are needed.

147.4.6.1 The XA Transaction Lifecycle

The transaction context for an XA transaction begins in the ACTIVE state, and may enter the MARKED_ROLLBACK state if the client calls setRollbackOnly().

An XA transaction must always return true from the `supportsXA()` method, indicating that XA participants may be registered using the `registerXAResource` method. XA transactions may also support one or more LocalResource participants. In this case the Transaction Context should also return `true` from the `supportsLocal()` method, indicating that LocalResource participants may be registered using the `registerLocalResource` method.

Once the transactional work has completed and the pre-completion callbacks have run the transaction should be completed using the normal XA algorithm. If the transaction fails during a commit attempt, resulting in a rollback, then the Transaction Control Service must generate a `TransactionRolledBackException`. If the transaction fails in any other way then the Transaction Control service must generate a `TransactionException`. Exceptions from the commit should be added to an existing ScopedWorkException if it exists.

147.4.6.2 XA Transaction Support Service Properties

A Transaction Control Service which supports XA transactions may be identified using the `osgi.xa.enabled` property which will be set to `Boolean.TRUE`.

If the Transaction Control Service also supports Local transactions then it must also set the `osgi.local.enabled` property to `Boolean.TRUE`.

147.5 Resource Providers

It is important that clients can easily control the transaction boundaries within their application, but it is equally important that the resources that the clients use participate in these transactions. In a Java EE Application server this is achieved by having the central application container create and manage all of the resources. In the Spring framework the Application context is responsible for ensuring that the resources are linked to a Transaction Manager.

There is no central container in OSGi, and so a modular solution is required. This specification defines the concept of a Resource Provider. A Resource Provider is a generic service which can provide a resource of some kind to the client. The Resource Provider exists to ensure that the resource being used will always be enlisted with the correct transaction context.

147.5.1 Generic Resource Providers

The purpose of a ResourceProvider is to provide the client with a configured resource which will automatically integrate with the correct transaction context at runtime.

Resources are created from a Resource Provider using the following method:

```
public <T> T getResource(TransactionControl txControl);
```

Typically clients will not use a plain Resource Provider, but will search for a specific subclass instead, which reifies the type parameter `T`. This allows for type safe access to resources, and ensures that the correct ResourceProvider implementation has been found.

147.5.1.1 The Basic Resource Lifecycle

Resources returned by a Resource Provider are proxies to an underlying factory for physical resources. Whenever the proxy is accessed then it should check the current transaction scope. If this is the first time the proxy has been accessed in the scope then the proxy should associate a new physical resource with the scope. If the scope is a Transaction scope then the resource must also be enlisted into the transaction at this point. Subsequent uses of the proxy within the same scope must use the same backing physical resource.

When a scope finishes any resources associated with the scope must be cleaned up without action required by the client. This rule applies to both the Transaction scope and the No Transaction scope, meaning that a client can safely write code using `TransactionControl#supports` without being concerned about resource leaks.

147.5.1.2 Unscoped Resource Access

If a resource is accessed by unscoped code then it must throw a TransactionException to indicate that it cannot be used without an active scope.

147.5.1.3 Closing, Flushing and Committing Resources

Most resources offer programmatic APIs for transaction and lifecycle management. For example `java.sql.Connection` has methods called `commit` and `close`.

If a client attempts to close a scoped resource then this operation should be silently ignored. The resource will be automatically cleaned up when the current scope completes and so there is no need to manually close the resource. Furthermore, if the resource were prematurely closed then it may prevent other services from accessing the resource within this scope.

If the resource is being used in a Transaction Scope then any transaction lifecycle methods, such as `commit` or `rollback`, must not delegate to the real resource and must throw a TransactionException instead.

147.5.1.4 Releasing Resource Providers

Resource Provider instances typically hold references to one or more physical resources, often in a pool. When a Resource Provider is no longer needed then it is important that these physical resources can be released to avoid resource leaks. The way in which a Resource Provider can determine it is no longer needed depends upon how the Resource Provider is created.

If the Resource Provider is registered directly as a service then it may release its physical resources when it is no longer used by any bundles. One way to achieve this is through the use of an OSGi Service Factory.

In some cases a Resource Provider is created by the client using a service from the service registry. In this case the lifecycle of the Resource Provider must be bounded by the lifecycle of the service that created it. In particular if the client bundle releases the service which created the Resource Provider then the Resource Provider must also be released. This mechanism ensures that Resource Providers do not need to be explicitly released by a client bundle when it stops. In addition services which create Resource Provider instances should provide a method which can be used to immediately release a particular Resource Provider instance without releasing service which created it. This allows client bundles to independently manage the lifecycle of multiple Resource Providers, and also to dynamically replace a Resource Provider instance.

Once a Resource Provider has been released then all proxy instances associated with it must be invalidated, and all methods on the proxies throw `TransactionException`.

147.5.2 JDBC Resource Providers

One of the most common resources to use in a transaction is a JDBC Connection. This specification therefore defines a specialized resource provider for obtaining JDBC Connections called a `JDBCConnectionProvider`. The purpose of this type is simply to reify the generic type of the ResourceProvider interface.

The scoped resource for a JDBC Connection Provider is a JDBC connection. The scoped resource allows for JDBC connections to be transparently pooled, enlisted in Transaction Scopes, and automatically closed.

147.5.2.1 JDBC and Transaction Scopes

When enlisted in an Active Transaction a JDBC connection will have autocommit set to false. Also the following methods must not be called by the client and must trigger a `TransactionException` if called.

- `commit()`
- `setAutoCommit()`
- `setSavepoint()`
- `setSavepoint(String)`
- `releaseSavepoint()`
- `rollback()`
- `rollback(Savepoint)`

If the Active Transaction commits the JDBC Connection must commit any work performed in the transaction. Similarly if the Active Transaction rolls back then the JDBC Connection must roll back any work performed in the transaction. After the transaction completes the JDBC connection must be cleaned up in an appropriate way, for example by closing it or returning it to a connection pool. There is no need for the client to close the connection, and any attempt to do so must be ignored by the resource provider.

147.5.2.2 JDBC and No Transaction Scopes

When accessed with from the No Transaction scope the JDBC connection may have autocommit set to true or false depending on the underlying configuration of the resource provider. This value may be changed by the client by using `setAutoCommit` within the scope, but the value will be reset after the end of the scope.

In the No Transaction context the JDBC connection will not be committed or rolled back by the Transaction Control Service or the Resource Provider. It is therefore the client's responsibility to call `commit` or `rollback` if appropriate. Savepoints may be used for partial rollback if desired.

After the end of the scope the JDBC connection must be automatically cleaned up by the Resource Provider in an appropriate way, for example by closing it or returning it to a connection pool. There is no need for the client to close the connection, and any attempt to do so must be ignored by the resource provider.

147.5.2.3 Closing the JDBC connection

As for all resource providers, calls to close() the connection must be ignored. JDBC connections also have an abort() method. Abort is effectively an asynchronous close operation for a JDBC connection, and so must also be ignored for any scoped connection.

147.5.2.4 The JDBCConnectionProviderFactory

The JDBCConnectionProvider may be provided as a service directly in the OSGi service registry, however this may not be acceptable in all use cases. JDBC Connections are often authenticated using a username and password. If the username and password relate to a specific bundle then it may not be appropriate to have the fully configured connections available in the Service Registry. In this case the JDBCConnectionProviderFactory offers several factory methods that can programmatically create a JDBCConnectionProvider.

147.5.2.4.1 JDBCConnectionProvider Configuration

Each factory method on the JDBCConnectionProviderFactory supplies set of properties which are used to configure the JDBCConnectionProvider, including the connection pooling behavior, and whether the ResourceProvider can be enlisted with XA and/or Local transactions.

By default the JDBCConnectionProvider will have a pool of 10 connections with a connection timeout of 30 seconds, an idle timeout of 10 minutes and a maximum connection lifetime of 3 hours. The JDBCConnectionProvider will also, by default, work all transaction types supported by the resource provider.

If the JDBCConnectionProvider is configured to enable XA then the DataSourceFactory being used must support XADataSource creation. If a pre-configured DataSource is supplied then it must be able to be unwrapped to an XADataSource.

147.5.2.4.2 Creating a JDBCConnectionProvider Using a DataSourceFactory

In this case the client provides the DataSourceFactory that should be used, along with the properties that should be used to create the DataSource/XADataSource. If XA transactions are enabled then the factory must create an XADataSource, otherwise the “osgi.use.driver” property can be used to force the creation of a Driver instance rather than a DataSource.

147.5.2.4.3 Creating a JDBCConnectionProvider Using a DataSource

In this case the client provides a pre-configured DataSource that should be used. If XA transactions are enabled then the DataSource must be able to be unwrapped to an XADataSource using the `unwrap` method.

147.5.2.4.4 Creating a JDBCConnectionProvider Using an XADataSource

In this case the client provides a preconfigured XADataSource that should be used by the resource provider.

147.5.2.4.5 Creating a JDBCConnectionProvider Using a Driver

In this case the client provides an instantiated driver class that should be used, along with the properties that should be used to create the JDBC connection. The JDBC properties must include a JDBC url to use when connecting to the database. XA transactions may not be enabled when using a Driver instance.

147.5.2.4.6 Releasing a JDBCConnectionProvider

In some cases a client of the JDBCConnectionProviderFactory may wish to release a created JDBCConnectionProvider without releasing the JDBCConnectionProviderFactory service. In this case the JDBCConnectionProvider instance should be passed to the `releaseProvider` method, which will immediately release the Resource Provider.

147.5.2.5 JDBCResourceProvider Examples

Setting up data Access with Declarative Services:

```
@Reference
TransactionControl txControl;

@Reference
JDBCConnectionProviderFactory resourceProviderFactory;

@Reference
DataSourceFactory dsf;

Connection connection;

@Activate
public void setUp(Config config) {
    Properties jdbc = new Properties();
    jdbc.setProperty(JDBC_URL, config.getURL());

    connection = resourceProviderFactory.getProviderFor(dsf, jdbc, null)
                    .getResource(txControl);
}
```

Reading data from a table:

```
txControl.supports(() -> {
        ResultSet rs = connection.createStatement()
                .executeQuery("Select message from TEST_TABLE");

        rs.next();
        return rs.getString(1);
    });
```

Updating a table:

```
txControl.required(() ->
        connection.createStatement()
           .execute("Insert into TEST_TABLE values ( 'Hello World!' )")
    );
```

147.5.3 JPA

JPA is a popular Object Relational Mapping (ORM) framework used to abstract away the low-level database access from business code. As an alternative means of accessing a database it is just as important for JPA resources to participate in transactions as it is for JDBC resources. This RFC therefore defines the JPAEntityManagerProvider interface as a specialized resource provider for JPA.

147.5.3.1 JPA and Transaction Scopes

When enlisted in a Transaction a JPA EntityManager will automatically track the state of persisted entity types and update the database as necessary. When participating in a transaction it is forbidden to call the getTransaction method on the EntityManager as manual transaction management is disabled. The joinTransaction method, however must be a no-op, and the isJoinedToTransaction must always return true.

If the Transaction commits the JPA EntityManager must commit any work performed in the transaction. Similarly if the Transaction rolls back then the JPA EntityManager must roll back any work performed in the transaction. After the transaction completes the JPA EntityManager must be cleaned up in an appropriate way, for example by closing it or returning it to a pool. There is no need for the client to close the entity manager, and any attempt to do so must be ignored by the resource provider.

147.5.3.2 JPA and No Transaction Scopes

When accessed with from the No Transaction scope the JPA EntityManager will not be participating in a Transaction or rolled back, it is therefore the client's responsibility to set up an EntityTransaction and to call commit or rollback as appropriate.

The joinTransaction method must throw a TransactionException, and the isJoinedToTransaction must always return false.

After the end of the scope the EntityManager must be automatically cleaned up in an appropriate way, for example by closing it or returning it to a pool.

147.5.3.3 RESOURCE_LOCAL and JTA EntityManagerFactory instances

When defining a JPA Persistence Unit the author must declare whether the EntityManagerFactory integrates with JTA transactions, or is suitable for resource local usage. The JPAEntityManagerProvider must take this into account when creating the transactional resource.

JTA scoped EntityManager instances may not manage their own transactions and must throw a JPA TransactionRequiredException if the client attempts to use the EntityTransaction interface. In effect the EntityManager behaves as a Synchronized, Transaction-Scoped, Managed Persistence Context as per the JPA 2.1 Specification. It is important to ensure that the Database connections used in a JTA Persistence Unit are integrated with the ongoing transaction.

RESOURCE_LOCAL scoped EntityManager instances may not participate in XA transactions, but otherwise behave in much the same way as JTA EntityManager instances. The one significant difference is that RESOURCE_LOCAL EntityManager instances may obtain an EntityTransaction when running in the No Transaction context.

147.5.3.4 The JPAEntityManagerProvider Factory

The JPAEntityManagerProvider may be provided directly in the OSGi service registry, however this may not be acceptable in all use cases. Database Connections are often authenticated using a username and password. If the username and password relate to a specific bundle then it may not be appropriate to have the configured connections available in the Service Registry. In this case the JPAEntityManagerProviderFactory offers several factory methods that can programmatically create a JPAEntityManagerProvider.

147.5.3.4.1 Creating a JPAEntityManagerProvider Using an EntityManagerFactoryBuilder

In this case the client provides the EntityManagerFactoryBuilder that should be used, along with the properties that should be used to create the EntityManagerFactory.

The typical reason for using an EntityManagerFactoryBuilder is to allow for the late binding of configuration, such as the database location. To support this usage pattern it is best to specify as few properties as possible inside the persistence descriptor. For example:

```
<persistence-unit name="test-unit">
  <description>My application's persistence unit</description>
</persistence-unit>
```

Passing String class names and expecting the JPA provider to load the Database driver reflectively should be avoided, however a configured DataSource can be passed using the javax.persistence.jtaDataSource property. If the JPA resource provider supports XA transactions then this property may be used to pass a configured XADataSource to be enlisted by the provider.

The osgi.jdbc.provider property can be passed to the resource provider defining a JDBCConnectionProvider that should be converted into a DataSource and passed to the EntityManageFactoryBuilder using the javax.persistence.jtaDataSource property. This allows the same physical database connection to be used by JPA and by JDBC within the same scope. Note that when using the osgi.jdbc.provider property to provide a connection to the database the JPA Resource Provider implementation should ignore configuration properties that cannot be acted upon, for example connection pool configuration, or setting an XA recovery identifier.

When configured to use JTA transactions most JPA implementations require integration with the transaction lifecycle. The JPA resource provider is required introspect the Entity Manager Factory Builder and to provide sufficient configuration to integrate the JPA provider with the supplied Transaction Control service. If the JPA resource provider is unable to supply the necessary configuration for the JPA implementation being used then it must log a warning.

147.5.3.4.2 Creating a JPAEntityManagerProvider Using an EntityManagerFactory

In this case the client provides the configured EntityManagerFactory that should be used, along with the properties that should be used to create the EntityManager.

When using an EntityManagerFactory to create the JPA resource provider there is no possibility for the resource provider implementation to customize the configuration of the EntityManagerFactory. This means that the client is responsible for fully configuring the EntityManagerFactory in this case. For Local Transactions this is reasonably simple, however for XA transactions this configuration process may be very involved. For example JPA providers typically require custom plugins to integrate with external Transaction lifecycle management. It is recommended that clients use the Entity Manager Factory Builder when XA transactions are needed.

147.5.3.4.3 Releasing a JPAEntityManagerProvider

In some cases a client of the JPAEntityManagerProviderFactory may wish to release a created JPAEntityManagerProvider without releasing the JPAEntityManagerProviderFactory service. In this case the JPAEntityManagerProvider instance should be passed to the releaseProvider method, which will immediately release the Resource Provider.

147.5.4 Connection Pooling

Database connections are usually heavyweight objects that require significant time to create. They may also consume physical resources such as memory or network ports. Creating a new database connection for every request is therefore wasteful, and adds unnecessary load to both the application and the database. Caching of database connections is therefore a useful way of improving performance. On the other hand applications must be careful not to create too many database connections. If one thousand requests arrive simultaneously then creating one thousand database connections is likely to crash the database server. These two requirements make database connections an excellent candidate for pooling. A small number of connections are made available and recycled after use. This saves the cost of recreating the connection and limits the overall load on the database.

In fact pooling is an excellent solution for many transactional resources, including JMS and EIS access.

147.5.4.1 Pooling in OSGi

Pooling has traditionally been difficult in OSGi because most connection pooling libraries use reflective access to load the underlying resource connector. This obviously fails unless the pooling library creates a static wiring to the connector, or has dynamic package imports. Both of these solutions are bad practices which create brittle dependencies.

The correct way to obtain Database connections in OSGi is to use a DataSourceFactory, however this offers no Connection Pooling. There is no real equivalent of a DataSourceFactory for JMS ConnectionFactory instances, but they also require manual decoration to enable connection pooling.

As pooling is such a core requirement for transactional resource access it is required for JDBCConnectionProviderFactory and JPAEntityManagerProviderFactory instances to offer connection pooling. The resource provider properties can be used to override the connection pooling configuration defaults (or to disable connection pooling entirely).

Third party resource providers should offer connection pooling using the same configuration properties and defaults wherever possible.

Table 147.4 *Pooling configuration properties*

Property Name	Default	Description
osgi.connection.pooling.enabled	true	Whether connection pooling is enabled for this ResourceProvider
osgi.connection.timeout	30000	The maximum time that a client will wait for a connection (in ms)
osgi.idle.timeout	180000	The time that a connection will remain idle before being closed (in ms)
osgi.connection.lifetime	10800000	The maximum time that a connection will remain open (in ms)
osgi.connection.min	10	The minimum number of connections that will be kept alive
osgi.connection.max	10	The maximum number of connections that will exist in the pool

147.6 Transaction Recovery

The XA transaction protocol defines a recovery mechanism which can be used to resolve in-doubt transactions. This is based upon the interaction of an XA Transaction Manager with an XAResource. In an OSGi environment resources may come and go at any time, as may Transaction Manager instances. Transaction recovery in OSGi is therefore a continuous, rather than a one-time process.

There are two main recovery scenarios that must be resolved by a Transaction Manager:

- Failure of one or more remote resources before the end of the transaction. In this case the Transaction Manager remains running and can roll-back or commit the other resources as appropriate. When the failed resource(s) eventually become available again the Transaction Manager can complete the in-doubt Transaction branch by committing it or rolling it back as appropriate.
- Failure of the Transaction Manager before the end of the transaction. In this case the Transaction Manager must use its recovery log to discover any in-doubt transaction branches. When the

resources associated with the in-doubt transaction branches become available the Transaction Manager can resolve the in-doubt branch by committing or rolling it back as appropriate.

In both of these cases it is crucial that the Transaction Manager can uniquely identify the resource that is being recovered. The Transaction Manager must be able to tell that a returning resource is suitable for recovering an in-doubt transaction branch.

The transaction branch itself has an Xid, which could theoretically be used to identify the resource. The problem with this, however, is that if the resource has already completed the transaction branch (for example if the failure occurred after sending a commit operation) then the resource may have discarded the Xid. We therefore require another identifier for a resource. The identifier must be unique to the Transaction Manager, but need not be Globally Unique. The identifier must also be persistent, that is, the same resource must have the same identifier after a restart of the OSGi framework. This ensures that transaction recovery can occur after a system crash.

147.6.1 Enlisting a Recoverable Resource in a Transaction

When a recoverable XA resource is associated with a TransactionContext using the `registerXAResource` method the resource identifier String is passed as a second argument. This is the identifier that will be used to locate the resource during recovery. If the XAResource is not recoverable then it may simply pass null as the second argument when registering.

147.6.2 Providing an XAResource for Recovery

When recovery is required the Transaction Manager may or may not be actively processing transactions involving the required recoverable resource. Therefore the Transaction Control service must be able to locate and obtain an XAResource instance for a named ResourceProvider.

To enable this the ResourceProvider must provide a whiteboard service which implements the `RecoverableXAResource` interface. This interface provides the resource identifier, and acts as a factory for XAResources that can be used to recover Transaction Branches.

The Transaction Control service can use this whiteboard to locate the correct XAResource to use. It may be, however, that when recovery is attempted it is not possible to provide a valid XAResource. In this case the RecoverableXAResource service may throw an exception. For example if the ResourceProvider is providing pooling and the pool is currently fully used then this may result in an exception.

Once the Transaction Control service has finished attempting to recover a Transaction branch then it must release the XAResource it obtained from the RecoverableXAResource using the `releaseXAResource` method.

147.6.3 Identifying implementations which support recovery

Transaction Control implementations which support recovery must register the Transaction Control service with the `osgi.recovery.enabled` service property with a value of `true` if recovery is enabled. Recovery may only be enabled if the implementation is configured for recovery, for example by configuring a transaction log.

Resource Provider factory services which support creating recoverable scoped resources must also register the `osgi.recovery.enabled` service property with a value of `true`. The recovery identifier of a scoped resource created by the factory is specified using the `osgi.recovery.identifier` property. It is an error to attempt to create a recoverable scoped resource from a factory which does not support recovery, and a `TransactionException` will be thrown to the caller if they attempt to set a recovery identifier when using a factory that does not support recovery.

147.7 Capabilities

Implementations of the Transaction Control Service specification must provide a capability in the osgi.service namespace representing the TransactionControl service. This capability must also declare a uses constraint for the org.osgi.service.transaction.control package, and attributes indicating whether the service supports local transactions, XA transactions, and recovery. For example, an XA capable, recoverable Transaction Control implementation which also supports recovery would offer the following capability.

```
Provide-Capability: osgi.service;objectClass:List<String>=
    "org.osgi.service.transaction.control.TransactionControl";
    uses:="org.osgi.service.transaction.control";osgi.local.enabled="true";
    osgi.xa.enabled="true";osgi.recovery.enabled="true"
```

Resource Provider Implementations must provide capabilities in the osgi.service namespace representing the ResourceProvider services and any factory services that they provide. These capabilities must also declare uses constraints for the org.osgi.service.transaction.control package and any other packages that they provide. In the case where a more specific type is registered (for example JDBCConnectionProvider) then that type should be used instead. The service properties that indicate whether the resource provider supports local transactions, XA transactions, and recovery must be advertised as attributes. For example:

```
Provide-Capability: osgi.service;objectClass:List<String>=
    "org.osgi.service.transaction.control.jdbc.JDBCConnectionProvider";
    uses:="org.osgi.service.transaction.control,org.osgi.service.transaction.
    control.jdbc";osgi.local.enabled="true";osgi.xa.enabled="true";
    osgi.recovery.enabled="true",
    osgi.service;objectClass:List<String>=
    "org.osgi.service.transaction.control.jdbc.JDBCConnectionProviderFactory";
    uses:="org.osgi.service.transaction.control,org.osgi.service.transaction.
    control.jdbc";osgi.local.enabled="true";osgi.xa.enabled="true";
    osgi.recovery.enabled="true"
```

These capabilities must follow the rules defined for the *osgi.service Namespace* on page 711.

147.8 Security

Access to the Transaction Control service and to Resource Provider services can be secured through the standard OSGi service permission model.

Clients should be aware that when they run scoped work there will be code from the Transaction Control service on the stack. Client operations that require specific privileges will therefore have to be performed inside a doPrivileged block.

147.9 org.osgi.service.transaction.control

Transaction Control Service Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.transaction.control; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.transaction.control; version="[1.0,1.1)"
```

147.9.1 Summary

- `LocalResource` - Resources that can integrate with local transactions should do so using this interface
- `ResourceProvider` - A resource provider is used to provide a transactional resource to the application
- `ScopedWorkException` - An Exception that is thrown when a piece of scoped work exits with an Exception.
- `TransactionBuilder` - A builder for a piece of transactional work
- `TransactionContext` - A transaction context defines the current transaction, and allows resources to register information and/or synchronizations
- `TransactionControl` - The interface used by clients to control the active transaction context
- `TransactionException` - An Exception indicating that there was a problem with starting, finishing, suspending or resuming a transaction
- `TransactionRolledBackException` - An Exception indicating that the active transaction was unexpectedly rolled back
- `TransactionStarter` - Implementations of this interface are able to run a piece of work within a transaction
- `TransactionStatus` - The status of the transaction A transaction may not enter all of the states in this enum, however it will always traverse the enum in ascending order.

147.9.2 public interface LocalResource

Resources that can integrate with local transactions should do so using this interface

147.9.2.1 public void commit() throws TransactionException

□ Commit the resource

Throws `TransactionException`–

147.9.2.2 public void rollback() throws TransactionException

□ Roll back the resource

Throws `TransactionException`–

147.9.3 public interface ResourceProvider<T>

<T>

A resource provider is used to provide a transactional resource to the application

147.9.3.1 public T getResource(TransactionControl txControl) throws TransactionException

txControl

□ Get a resource which will associate with the current transaction context when used

Returns The resource which will participate in the current transaction

Throws `TransactionException`– if the resource cannot be registered with the transaction

147.9.4 public class ScopedWorkException extends RuntimeException

An Exception that is thrown when a piece of scoped work exits with an Exception.

If the scope was inherited and therefore is still active when this exception is raised then the current TransactionContext will be available from the ongoingContext() method.

Provider Type Consumers of this API must not implement this type

147.9.4.1 public ScopedWorkException(String message, Throwable cause, TransactionContext context)

message

cause

context

□ Creates a new TransactionException with the supplied message and cause

147.9.4.2 public T extends Throwable as(Class<T> throwable) throws T

Type Parameters `<T extends Throwable>`

throwable

□ Throws the cause of this Exception as a RuntimeException the supplied Exception type.

Usage is of the form:

```
public void doStuff() throws IOException {
    try {
        ...
    } catch (ScopedWorkException swe) {
        throw swe.as(IOException.class);
    }
}
```

Returns This method will always throw an exception

Throws T–

147.9.4.3 public RuntimeException asOneOf(Class<A> a, Class<B> b) throws A, B

Type Parameters `<A extends Throwable, B extends Throwable>`

a

b

□ Throws the cause of this Exception as a RuntimeException or one of the supplied Exception types.

Usage is of the form:

```
public void doStuff() throws IOException, ClassNotFoundException {
    try {
        ...
    } catch (ScopedWorkException swe) {
        throw swe.asOneOf(IOException.class, ClassNotFoundException.class);
    }
}
```

Returns This method will always throw an exception

Throws A–

B–

147.9.4.4 public RuntimeException asOneOf(Class<A> a, Class<B> b, Class<C> c) throws A, B, C

Type Parameters `<A extends Throwable, B extends Throwable, C extends Throwable>`

a

b

c

□ Throws the cause of this Exception as a RuntimeException or one of the supplied Exception types.

Returns This method will always throw an exception

Throws A–

B–

See Also asOneOf(Class, Class)

147.9.4.5 public RuntimeException asOneOf(Class<A> a, Class<B> b, Class<C> c, Class<D> d) throws A, B, C, D

Type Parameters `<A extends Throwable, B extends Throwable, C extends Throwable, D extends Throwable>`

a

b

c

d

□ Throws the cause of this Exception as a RuntimeException or one of the supplied Exception types.

Returns This method will always throw an exception

Throws A–

B–

C–

D–

See Also asOneOf(Class, Class)

147.9.4.6 public RuntimeException asRuntimeException()

Returns The cause of this Exception as a RuntimeException if it is one, or this otherwise

147.9.4.7 public TransactionContext ongoingContext()

Returns The ongoing transaction context if the current scope was still active when this exception was raised or null otherwise. Note that this property will not be persisted during serialization.

147.9.5 public abstract class TransactionBuilder implements TransactionStarter

A builder for a piece of transactional work

Provider Type Consumers of this API must not implement this type

147.9.5.1 protected final List<Class<? extends Throwable>> noRollbackFor

The list of Throwable types that must not trigger rollback

147.9.5.2 protected final List<Class<? extends Throwable>> rollbackFor

The list of Throwable types that must trigger rollback

147.9.5.3 public TransactionBuilder()

147.9.5.4 public final TransactionBuilder noRollbackFor(Class<? extends Throwable> t, Class<? extends Throwable>... throwables)

t An exception type that should not trigger rollback

throwables further exception types that should not trigger rollback

□ Declare a list of Exception types (and their subtypes) that *must not* trigger a rollback. By default the transaction will rollback for all Exceptions. If an Exception type is registered using this method then that type and its subtypes will *not* trigger rollback. If the same type is registered using both rollbackFor(Class, Class...) and noRollbackFor(Class, Class...) then the transaction *will not* begin and will instead throw a TransactionException

Note that the behavior of this method differs from Java EE and Spring in two ways:

- In Java EE and Spring transaction management checked exceptions are considered "normal returns" and do not trigger rollback. Using an Exception as a normal return value is considered a *bad* design practice. In addition this means that checked Exceptions such as java.sql.SQLException do not trigger rollback by default. This, in turn, leads to implementation mistakes that break the transactional behavior of applications.
- In Java EE it is legal to specify the same Exception type in rollbackFor and noRollbackFor. Stating that the same Exception should both trigger *and* not trigger rollback is a logical impossibility, and clearly indicates an API usage error. This API therefore enforces usage by triggering an exception in this invalid case.

Returns this builder

147.9.5.5 public abstract TransactionBuilder readOnly()

□ Indicate to the Transaction Control service that this transaction will be read-only. This hint may be used by the Transaction Control service and associated resources to optimize the transaction.

Note that this method is for optimization purposes only. The TransactionControl service is free to ignore the call if it does not offer read-only optimization.

If a transaction is marked read-only and then the scoped work performs a write operation on a resource then this is a programming error. The resource is free to raise an exception when the write is attempted, or to permit the write operation. As a result the transaction may commit successfully, or may rollback.

Returns this builder

147.9.5.6 public final TransactionBuilder rollbackFor(Class<? extends Throwable> t, Class<? extends Throwable>... throwables)

t

throwables The Exception types that should trigger rollback

□ Declare a list of Exception types (and their subtypes) that *must* trigger a rollback. By default the transaction will rollback for all Exceptions. If a more specific type is registered using noRollbackFor(Class, Class...) then that type will not trigger rollback. If the same type is registered using both rollbackFor(Class, Class...) and noRollbackFor(Class, Class...) then the transaction *will not* begin and will instead throw a TransactionException

Note that the behavior of this method differs from Java EE and Spring in two ways:

- In Java EE and Spring transaction management checked exceptions are considered "normal returns" and do not trigger rollback. Using an Exception as a normal return value is considered a *bad* design practice. In addition this means that checked Exceptions such as java.sql.SQLException do not trigger rollback by default. This, in turn, leads to implementation mistakes that break the transactional behavior of applications.

- In Java EE it is legal to specify the same Exception type in rollbackFor and noRollbackFor. Stating that the same Exception should both trigger *and* not trigger rollback is a logical impossibility, and clearly indicates an API usage error. This API therefore enforces usage by triggering an exception in this invalid case.

Returns this builder

147.9.6 public interface TransactionContext

A transaction context defines the current transaction, and allows resources to register information and/or synchronizations

Provider Type Consumers of this API must not implement this type

147.9.6.1 public boolean getRollbackOnly() throws IllegalStateException

□ Is this transaction marked for rollback only

Returns true if this transaction is rollback only

Throws IllegalStateException – if no transaction is active

147.9.6.2 public Object getScopedValue(Object key)

key

□ Get a value scoped to this transaction

Returns The resource, or null

147.9.6.3 public Object getTransactionKey()

□ Get the key associated with the current transaction

Returns the transaction key, or null if there is no transaction

147.9.6.4 public TransactionStatus getTransactionStatus()

Returns The current transaction status

147.9.6.5 public boolean isReadOnly()

Returns true if the TransactionContext supports read-only optimizations *and* the transaction was marked read only. In particular it is legal for this method to return false even if the transaction was marked read only by the initiating client.

147.9.6.6 public void postCompletion(Consumer<TransactionStatus> job) throws IllegalStateException

job

□ Register a callback that will be made after the scope completes

For transactional scopes the state of the scope will be either TransactionStatus.COMMITTED or TransactionStatus.ROLLED_BACK.

For no-transaction scopes the state of the scope will always be TransactionStatus.NO_TRANSACTION.

Post-completion callbacks should not throw Exceptions and cannot affect the outcome of a piece of scoped work

Throws IllegalStateException– if no transaction is active

147.9.6.7 public void preCompletion(Runnable job) throws IllegalStateException

job The action to perform before completing the scope

□ Register a callback that will be made before a scope completes.

For transactional scopes the state of the scope will be either TransactionStatus.ACTIVE or TransactionStatus.MARKED_ROLLBACK. Pre-completion callbacks may call setRollbackOnly() to prevent a commit from proceeding.

For no-transaction scopes the state of the scope will always be TransactionStatus.NO_TRANSACTION.

Exceptions thrown by pre-completion callbacks are treated as if they were thrown by the scoped work, including any configured commit or rollback behaviors for transactional scopes.

Throws IllegalStateException– if the transaction has already passed beyond the TransactionStatus.MARKED_ROLLBACK state

147.9.6.8 public void putScopedValue(Object key, Object value)

key

value

□ Associate a value with this transaction

147.9.6.9 public void registerLocalResource(LocalResource resource) throws IllegalStateException

resource

□ Register a Local resource with the current transaction

Throws IllegalStateException– if no transaction is active, or the current transaction does not support local resources.

147.9.6.10 public void registerXAResource(XAResource resource, String recoveryId) throws IllegalStateException

resource

recoveryId The resource id to be used for recovery, the id may be null if this resource is not recoverable.

If an id is passed then a RecoverableXAResource with the same id must be registered in the service registry for recovery to occur.

If the underlying TransactionControl service does not support recovery then it must treat the resource as if it is not recoverable.

□ Register an XA resource with the current transaction

Throws IllegalStateException– if no transaction is active, or the current transaction is not XA capable

147.9.6.11 public void setRollbackOnly() throws IllegalStateException

□ Mark this transaction for rollback

Throws IllegalStateException– if no transaction is active

147.9.6.12 public boolean supportsLocal()

Returns true if the current transaction supports Local resources

147.9.6.13 public boolean supportsXA()

Returns true if the current transaction supports XA resources

147.9.7 public interface TransactionControl extends TransactionStarter

The interface used by clients to control the active transaction context

Provider Type Consumers of this API must not implement this type

147.9.7.1 public boolean activeScope()

Returns true if a transaction is currently active, or if there is a "no transaction" context active

147.9.7.2 public boolean activeTransaction()

Returns true if a transaction is currently active

147.9.7.3 public TransactionBuilder build()

□ Build a transaction context to surround a piece of transactional work

Returns A builder to complete the creation of the transaction

147.9.7.4 public TransactionContext getCurrentContext()

Returns The current transaction context, which may be a "no transaction" context, or null if there is no active context

147.9.7.5 public boolean getRollbackOnly() throws IllegalStateException

□ Gets the rollback status of the active transaction

Returns true if the transaction is marked for rollback

Throws `IllegalStateException` – if no transaction is active

147.9.7.6 public void ignoreException(Throwable t) throws IllegalStateException

t The exception to ignore

□ Marks that the current transaction should not be rolled back if the supplied Exception is thrown by the current transactional work

Throws `IllegalStateException` – if no transaction is active

147.9.7.7 public void setRollbackOnly() throws IllegalStateException

□ Marks the current transaction to be rolled back

Throws `IllegalStateException` – if no transaction is active

147.9.8 public class TransactionException extends RuntimeException

An Exception indicating that there was a problem with starting, finishing, suspending or resuming a transaction

Provider Type Consumers of this API must not implement this type

147.9.8.1 public TransactionException(String message)

message

□ Creates a new TransactionException with the supplied message

147.9.8.2 public TransactionException(String message, Throwable cause)

message

cause

- Creates a new TransactionException with the supplied message and cause

147.9.9 public class TransactionRolledBackException extends TransactionException

An Exception indicating that the active transaction was unexpectedly rolled back

Provider Type Consumers of this API must not implement this type

147.9.9.1 public TransactionRolledBackException(String message)

message

- Create a new TransactionRolledBackException with the supplied message

147.9.9.2 public TransactionRolledBackException(String message, Throwable cause)

cause

message

- Create a new TransactionRolledBackException with the supplied message

147.9.10 public interface TransactionStarter

Implementations of this interface are able to run a piece of work within a transaction

Provider Type Consumers of this API must not implement this type

147.9.10.1 public T notSupported(Callable<T> work) throws TransactionException, ScopedWorkException

Type Parameters <T>

work

- The supplied piece of work must be run outside the context of a transaction. If an existing transaction is active then it must be suspended and a "no transaction" context associated with the work. After the work has completed any suspended transaction must be resumed.

  The "no transaction" context does not support resource enlistment, and will not commit or rollback any changes, however it does provide a post completion callback to any registered functions. This function is suitable for final cleanup, such as closing a connection

Returns The value returned by the work

Throws `TransactionException`– if there is an error starting or completing the transaction

`ScopedWorkException`– if the supplied work throws an Exception

147.9.10.2 public T required(Callable<T> work) throws TransactionException, TransactionRolledBackException, ScopedWorkException

Type Parameters <T>

work

- A transaction is required to run the supplied piece of work. If no transaction is active then it must be started and associated with the work and then completed after the transactional work has finished.

Returns The value returned by the work

Throws `TransactionException`– if there is an error starting or completing the transaction

`TransactionRolledBackException`– if the transaction rolled back due to a failure in one of the resources or an internal error in the TransactionControl service

`ScopedWorkException`– if the supplied work throws an Exception

147.9.10.3 public T requiresNew(Callable<T> work) throws TransactionException, TransactionRolledBackException, ScopedWorkException

Type Parameters <T>

work

- A new transaction is required to run the supplied piece of work. If an existing transaction is active then it must suspended and a new transaction started and associated with the work. After the work has completed the new transaction must also complete and any suspended transaction be resumed.

Returns The value returned by the work

Throws `TransactionException`– if there is an error starting or completing the transaction

`TransactionRolledBackException`– if the transaction rolled back due to a failure

`ScopedWorkException`– if the supplied work throws an Exception

147.9.10.4 public T supports(Callable<T> work) throws TransactionException, ScopedWorkException

Type Parameters <T>

work

- The supplied piece of work may run inside or outside the context of a transaction. If an existing transaction or "no transaction" context is active then it will continue, otherwise a new "no transaction" context is associated with the work. After the work has completed any created transaction context must be completed.

  The "no transaction" context does not support resource enlistment, and will not commit or rollback any changes, however it does provide a post completion callback to any registered functions. This function is suitable for final cleanup, such as closing a connection

Returns The value returned by the work

Throws `TransactionException`– if there is an error starting or completing the transaction

`ScopedWorkException`– if the supplied work throws an Exception

147.9.11 enum TransactionStatus

The status of the transaction A transaction may not enter all of the states in this enum, however it will always traverse the enum in ascending order. In particular if the TransactionStatus is reported as X then it will never proceed into a state Y where X.compareTo(Y) >= 0;

147.9.11.1 NO_TRANSACTION

No transaction is currently active

147.9.11.2 ACTIVE

A transaction is currently in progress

147.9.11.3 MARKED_ROLLBACK

A transaction is currently in progress and has been marked for rollback

147.9.11.4 PREPARING

A two phase commit is occurring and the transaction is being prepared

147.9.11.5 PREPARED

A two phase commit is occurring and the transaction has been prepared

147.9.11.6 COMMITTING

The transaction is in the process of being committed

147.9.11.7 COMMITTED

The transaction has committed

147.9.11.8 ROLLING_BACK

The transaction is in the process of rolling back

147.9.11.9 ROLLED_BACK

The transaction has been rolled back

147.9.11.10 public static TransactionStatus valueOf(String name)

147.9.11.11 public static TransactionStatus[] values()

147.10 org.osgi.service.transaction.control.jdbc

Transaction Control JDBC Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.transaction.control.jdbc; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.transaction.control.jdbc; version="[1.0,1.1)"
```

147.10.1 Summary

- JDBCConnectionProvider - A specialized ResourceProvider suitable for obtaining JDBC connections.
- JDBCConnectionProviderFactory - A factory for creating JDBCConnectionProvider instances

147.10.2 public interface JDBCConnectionProvider extends ResourceProvider<Connection>

A specialized ResourceProvider suitable for obtaining JDBC connections.

Instances of this interface may be available in the Service Registry, or can be created using a JDBCConnectionProviderFactory.

147.10.3 public interface JDBCConnectionProviderFactory

A factory for creating JDBCConnectionProvider instances

This factory can be used if the JDBCConnectionProvider should not be a public service, for example to protect a username/password.

Provider Type Consumers of this API must not implement this type

147.10.3.1 public static final String CONNECTION_LIFETIME = "osgi.connection.lifetime"

The property used to set the maximum amount of time that connections in the pool should remain open

147.10.3.2 **public static final String CONNECTION_POOLING_ENABLED = "osgi.connection.pooling.enabled"**

The property used to determine whether connection pooling is enabled for this resource provider

147.10.3.3 **public static final String CONNECTION_TIMEOUT = "osgi.connection.timeout"**

The property used to set the maximum amount of time that the pool should wait for a connection

147.10.3.4 **public static final String IDLE_TIMEOUT = "osgi.idle.timeout"**

The property used to set the maximum amount of time that connections in the pool should remain idle before being closed

147.10.3.5 **public static final String LOCAL_ENLISTMENT_ENABLED = "osgi.local.enabled"**

The property used to determine whether local enlistment is enabled for this resource provider

147.10.3.6 **public static final String MAX_CONNECTIONS = "osgi.connection.max"**

The property used to set the maximum number of connections that should be held in the pool

147.10.3.7 **public static final String MIN_CONNECTIONS = "osgi.connection.min"**

The property used to set the minimum number of connections that should be held in the pool

147.10.3.8 **public static final String OSGI_RECOVERY_IDENTIFIER = "osgi.recovery.identifier"**

The property used to set the recovery identifier that should be used by this resource

147.10.3.9 **public static final String USE_DRIVER = "osgi.use.driver"**

The property used to set the maximum number of connections that should be held in the pool

147.10.3.10 **public static final String XA_ENLISTMENT_ENABLED = "osgi.xa.enabled"**

The property used to determine whether XA enlistment is enabled for this resource provider

147.10.3.11 **public static final String XA_RECOVERY_ENABLED = "osgi.recovery.enabled"**

The property used to determine whether XA recovery is enabled for this resource provider

147.10.3.12 **public JDBCConnectionProvider getProviderFor(DataSourceFactory dsf, Properties jdbcProperties, Map<String, Object> resourceProviderProperties)**

dsf

jdbcProperties The properties to pass to the DataSourceFactory in order to create the underlying DataSource

resourceProviderProperties Configuration properties to pass to the JDBC Resource Provider runtime

□ Create a private JDBCConnectionProvider using a DataSourceFactory. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

Returns A JDBCConnectionProvider that can be used in transactions

147.10.3.13 **public JDBCConnectionProvider getProviderFor(DataSource ds, Map<String, Object> resourceProviderProperties)**

ds

resourceProviderProperties Configuration properties to pass to the JDBC Resource Provider runtime

□ Create a private JDBCConnectionProvider using an existing DataSource. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

Returns A JDBCConnectionProvider that can be used in transactions

147.10.3.14 public JDBCConnectionProvider getProviderFor(Driver driver, Properties jdbcProperties, Map<String, Object> resourceProviderProperties)

driver

jdbcProperties The properties to pass to the Driver in order to create a Connection

resourceProviderProperties Configuration properties to pass to the JDBC Resource Provider runtime

□ Create a private JDBCConnectionProvider using an existing Driver. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

Returns A JDBCConnectionProvider that can be used in transactions

147.10.3.15 public JDBCConnectionProvider getProviderFor(XADataSource ds, Map<String, Object> resourceProviderProperties)

ds

resourceProviderProperties Configuration properties to pass to the JDBC Resource Provider runtime

□ Create a private JDBCConnectionProvider using an existing XADataSource. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

Returns A JDBCConnectionProvider that can be used in transactions

147.10.3.16 public void releaseProvider(JDBCConnectionProvider provider)

provider

□ Release a JDBCConnectionProvider instance that has been created by this factory. Released instances are eligible to be shut down and have any remaining open connections closed.

Note that all JDBCConnectionProvider instances created by this factory service are implicitly released when the factory service is released by this bundle.

Throws IllegalArgumentException– if the supplied resource was not created by this factory service instance.

147.11 org.osgi.service.transaction.control.jpa

Transaction Control JPA Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.transaction.control.jpa; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.transaction.control.jpa; version="[1.0,1.1)"
```

147.11.1 Summary

- JPAEntityManagerProvider - A specialized ResourceProvider suitable for obtaining JPA EntityManager instances.
- JPAEntityManagerProviderFactory - A factory for creating JPAEntityManagerProvider instances

147.11.2 public interface JPAEntityManagerProvider extends ResourceProvider<EntityManager>

A specialized ResourceProvider suitable for obtaining JPA EntityManager instances.

Instances of this interface may be available in the Service Registry, or can be created using a JPAEntityManagerProviderFactory.

147.11.3 public interface JPAEntityManagerProviderFactory

A factory for creating JPAEntityManagerProvider instances

This factory can be used if the JPAEntityManagerProvider should not be a public service, for example to protect a username/password.

Provider Type Consumers of this API must not implement this type

147.11.3.1 public static final String CONNECTION_LIFETIME = "osgi.connection.lifetime"

The property used to set the maximum amount of time that connections in the pool should remain open

147.11.3.2 public static final String CONNECTION_POOLING_ENABLED = "osgi.connection.pooling.enabled"

The property used to determine whether connection pooling is enabled for this resource provider

147.11.3.3 public static final String CONNECTION_TIMEOUT = "osgi.connection.timeout"

The property used to set the maximum amount of time that the pool should wait for a connection

147.11.3.4 public static final String IDLE_TIMEOUT = "osgi.idle.timeout"

The property used to set the maximum amount of time that connections in the pool should remain idle before being closed

147.11.3.5 public static final String LOCAL_ENLISTMENT_ENABLED = "osgi.local.enabled"

The property used to determine whether local enlistment is enabled for this resource provider

147.11.3.6 public static final String MAX_CONNECTIONS = "osgi.connection.max"

The property used to set the maximum number of connections that should be held in the pool

147.11.3.7 public static final String MIN_CONNECTIONS = "osgi.connection.min"

The property used to set the minimum number of connections that should be held in the pool

147.11.3.8 public static final String OSGI_RECOVERY_IDENTIFIER = "osgi.recovery.identifier"

The property used to set the recovery identifier that should be used by this resource

147.11.3.9 public static final String PRE_ENLISTED_DB_CONNECTION = "osgi.jdbc.enlisted"

The property used to indicate that database connections will be automatically enlisted in ongoing transactions without intervention from the JPA resource provider

147.11.3.10 public static final String TRANSACTIONAL_DB_CONNECTION = "osgi.jdbc.provider"

The property used to provide a JDBCConnectionProvider to the resource provider. This will be converted into a DataSource by the factory, and passed to the EntityManagerFactoryBuilder using the javax.persistence.jtaDataSource property

147.11.3.11 public static final String XA_ENLISTMENT_ENABLED = "osgi.xa.enabled"

The property used to determine whether XA enlistment is enabled for this resource provider

147.11.3.12 public static final String XA_RECOVERY_ENABLED = "osgi.recovery.enabled"

The property used to determine whether XA recovery is enabled for this resource provider

147.11.3.13 public JPAEntityManagerProvider getProviderFor(EntityManagerFactoryBuilder emfb, Map<String, Object> jpaProperties, Map<String, Object> resourceProviderProperties)

emfb

jpaProperties The properties to pass to the EntityManagerFactoryBuilder in order to create the underlying EntityManagerFactory and EntityManager instances

resourceProviderProperties Configuration properties to pass to the JPA Resource Provider runtime

□ Create a private JPAEntityManagerProvider using an EntityManagerFactoryBuilder. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

If XA transactions are used then this factory will provide configuration to ensure that the JPA Provider can participate correctly in ongoing transactions.

Returns A JPAEntityManagerProvider that can be used in transactions

147.11.3.14 public JPAEntityManagerProvider getProviderFor(EntityManagerFactory emf, Map<String, Object> resourceProviderProperties)

emf

resourceProviderProperties Configuration properties to pass to the JDBC Resource Provider runtime

□ Create a private JPAEntityManagerProvider using an existing EntityManagerFactory. This call may fail with a TransactionException if the supplied configuration is invalid. Examples of invalid configuration include:

- The properties request XA enlistment, but the provider implementation only supports local enlistment
- The properties attempt to set a recovery alias, but the provider does not support recovery.

When using this method the client is responsible for all configuration of the EntityManagerFactory. This includes setting any relevant integration plugins for ensuring that the JPA provider can participate in the ongoing transaction context.

Returns A JPAEntityManagerProvider that can be used in transactions

147.11.3.15 public void releaseProvider(JPAEntityManagerProvider provider)

provider

□ Release a JPAEntityManagerProvider instance that has been created by this factory. Released instances are eligible to be shut down and have any remaining open connections closed.

Note that all JPAEntityManagerProvider instances created by this factory service are implicitly released when the factory service is released by this bundle.

Throws `IllegalArgumentException`– if the supplied resource was not created by this factory service instance.

147.12 org.osgi.service.transaction.control.recovery

Transaction Control Service Recovery Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.transaction.control.recovery; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.transaction.control.recovery; version="[1.0,1.1)"`

147.12.1 Summary

- `RecoverableXAResource` - A RecoverableXAResource service may be provided by a ResourceProvider if they are able to support XA recovery operations.

147.12.2 public interface RecoverableXAResource

A RecoverableXAResource service may be provided by a ResourceProvider if they are able to support XA recovery operations. There are two main sorts of recovery:

- Recovery after a remote failure, where the local transaction manager runs throughout
- Recovery after a local failure, where the transaction manager replays in-doubt transactions from its log

This service is used in both of these cases. The identifier returned by getId() provides a persistent name that can be used to correlate usage of the resource both before and after failure. This identifier must also be passed to TransactionContext.registerXAResource(XAResource, String) each time the recoverable resource is used.

147.12.2.1 public static final String OSGI_RECOVERY_ENABLED = "osgi.recovery.enabled"

This service property key is used by TransactionControl services and ResourceProvider factories to indicate that they can support transaction recovery.

147.12.2.2 public String getId()

□ Get the id of this resource. This should be unique, and persist between restarts

Returns an identifier, never null

147.12.2.3 public XAResource getXAResource() throws Exception

□ Get a new, valid XAResource that can be used in recovery This XAResource will be returned later using the releaseXAResource(XAResource) method

Returns a valid, connected, XAResource

Throws Exception– If it is not possible to acquire a valid XAResource at the current time, for example if the database is temporarily unavailable.

147.12.2.4 public void releaseXAResource(XAResource xaRes)

xaRes An XAResource previously returned by getXAResource()

□ Release the XAResource that has been used for recovery

148 Cluster Information Specification

Version 1.0

148.1 Introduction

Modern enterprise applications are most often deployed on distributed infrastructure such as a private or public cloud environment, instead of on a single server. This is done to distribute the application load, to replicate the application to guarantee availability or to exploit dedicated hardware for certain application functionality (for example, a database server).

The unit of management is often no longer a single physical machine. Server infrastructure is nowadays mostly offered in a virtualized fashion, such as hardware virtualization using a hypervisor or operating system virtualization using containers. Potentially these can also be hierarchically managed, for example having multiple containers running inside a virtual machine. Therefore, it becomes key to manage an application running on a cluster of such (virtual) machines and/or containers.

Also in the context of the Internet of Things (IoT), often a number of gateway devices is deployed in the network that connect various sensors and actuators creating a smart environment. Again, it becomes key to discover and manage these devices, and query their capabilities.

The OSGi specification already provides chapters describing how to deploy software on remote infrastructure, how to call remote services or how manage a remote OSGi framework. In this chapter we define an API for a management agent to discover, list and inspect available nodes in the cluster.

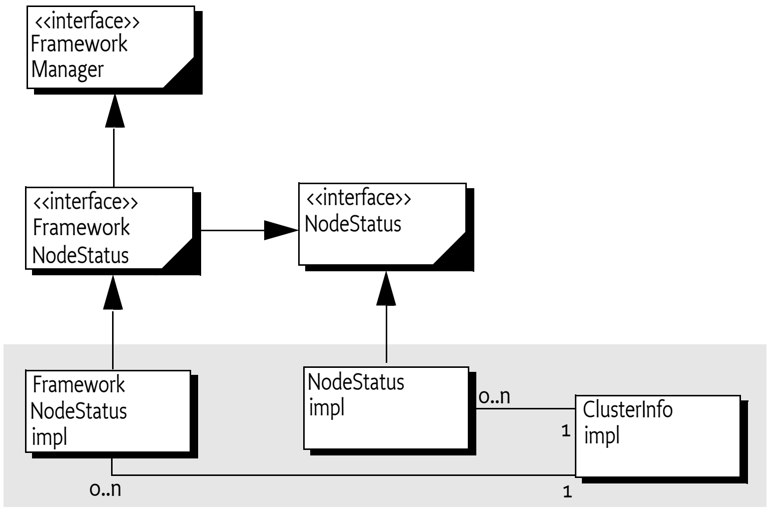
148.1.1 Essentials

- *Cluster* - A cluster is a collection of nodes connected by a network. Most often the nodes are managed by a public or private cloud provider.
- *Node* - A node is a discoverable entity in the cluster, for example a physical machine, a virtual machine (VM), a container or an OSGi framework.

148.1.2 Entities

- *NodeStatus* - The Node Status service represents a node in the cluster. This can be any entity in the cluster such as a database server, a virtual machine, a container, an OSGi framework, etc.
- *FrameworkNodeStatus* - The Framework Node Status service represents an OSGi framework in the cluster.
- *FrameworkManager* - The FrameworkManager service provides an interface to manage an OSGi framework in the cluster.

Figure 148.1 Clusterinfo Entity overview


<<interface>> Framework Manager
<<interface>> Framework NodeStatus
<<interface>> NodeStatus
Framework NodeStatus impl
NodeStatus impl
ClusterInfo impl
0..n
1
0..n
1

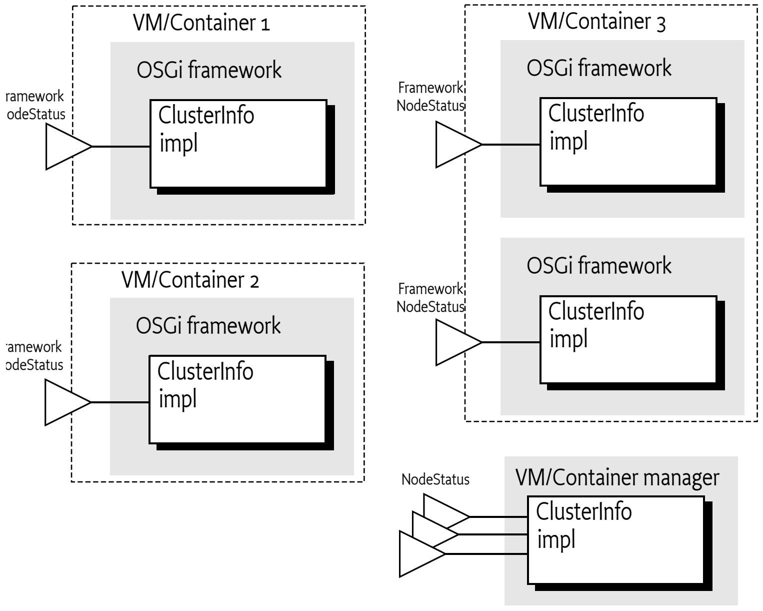
148.2 OSGi frameworks in a cluster

We distinguish two types of nodes in a cluster. On the one hand we have OSGi frameworks, which publish their presence using a Framework Node Status service. On the other hand there can be other nodes in the cluster, such as the virtual machines or containers the OSGi frameworks are running on, or an external server such as a database. These can be represented using a Node Status service.

When an OSGi framework is part of a cluster, this means it gets access to remote services of any other OSGi framework in that cluster. Ensuring the discovery, visibility and access of remote services within the cluster is the responsibility of the *Remote Service Admin Service Specification* on page 493.

An example cluster deployment is shown in Figure 148.2 on page 1005. Here, a cluster consisting of three virtual machines or containers has deployed a total of four OSGi frameworks. Each OSGi framework has a Cluster Information implementation running that exposes a Framework Node Status service. Besides these, there can also be an entity managing the virtual machines/containers (for example, the cloud provider), that exposes three Node Status services, one for each VM/container. In this case, each Framework Node Status will have a parent id pointing to the id of the Node Status of the VM/container it is deployed on.

Figure 148.2 Example cluster deployment



148.3 Node Status Service

The NodeStatus service advertises the availability of a node in the cluster. This node can be any entity in the cluster such as a physical machine, a virtual machine, a container or an OSGi framework.

The Node Status service provides metadata about the node via its service properties. Each Node Status must provide an id and cluster name. Optionally additional service properties can be provided such as the physical location of the node, the endpoints at which this node can be accessed, etc. These service properties can be converted to a NodeStatusDTO to have type-safe access to these properties using the *Converter Specification* on page 1457.

Table 148.1 Service properties of the NodeStatus service

Service Property Name	Type	Description
osgi.clusterinfo.id	String	The globally unique ID for this node. For example the Docker ID if this node is a Docker container, or the framework UUID if this node is an OSGi framework.
osgi.clusterinfo.cluster	String	The name of the cluster this node belongs to.
osgi.clusterinfo.parent	String	In the case this node is part of or embedded in another node, this field contains the id of the parent node. For example multiple virtual machines could run on the same physical node.
osgi.clusterinfo.endpoint	String+	The endpoint(s) at which this node can be accessed from the viewpoint of the consumer of the service.
osgi.clusterinfo.endpoint.private	String+	Private endpoint(s) at which this node can be accessed from within the cluster only.
osgi.clusterinfo.vendor	String	The vendor name of the cloud/environment in which the node operates.

Service Property Name	Type	Description
osgi.clusterinfo.version	String	The version of the cloud/environment in which the node operates. The value follows the versioning scheme of the cloud provider and may therefore not comply with the OSGi versioning syntax.
osgi.clusterinfo.country	String	ISO 3166-1 alpha-3 location where this node is running, if known.
osgi.clusterinfo.location	String	ISO 3166-2 location where this node is running, if known. This location is more detailed than the country code as it may contain province or territory.
osgi.clusterinfo.region	String	Something smaller than a country and bigger than a location (for example, us-east-1 or other cloud-specific location)
osgi.clusterinfo.zone	String	Regions are often subdivided in zones that represent different physical locations. The zone can be provided here.
osgi.clusterinfo.tags	String+	Tags associated with this node that can be contributed to by the provider and also by bundles.

The Node Status service can also provide access to some dynamic properties of the node. The getMetrics method allows to query key-value pairs, that are specific for that node. For example, for an OSGi framework these could be CPU and memory usage, for a database node these could be the number of database reads and writes, and for a VM these could be metrics made accessible by the cloud provider. In this case the service implementor can provide DTOs to have a type-safe way to access these metrics by converting the returned map to one of these DTOs. For example, an implementation could expose JMX metrics together with a type-safe DTO:

```
public class JMXMetricsDTO extends DTO {
  /**
   * The number of processors available
   */
  public int availableProcessors;

  /**
   * The average system load
   */
  public float systemLoadAverage;

  /**
   * The maximal amount of heap memory available to the JVM
   */
  public long heapMemoryMax;

  /**
   * The amount of heap memory used by the JVM
   */
  public long heapMemoryUsed;

  /**
   * The maximal amount of non-heap memory available to the JVM
   */
  public long nonHeapMemoryMax;

  /**
```

```
    * The amount of non-heap memory used by the JVM
    */
   public long nonHeapMemoryUsed;
}
```

Such DTO can be used to obtain metrics from a NodeStatus service as follows:

```
// From service registry
NodeStatus ns = ...;
// Obtain all metrics for this node
Map<String, Object> metrics = ns.getMetrics();

// Convert the metrics map to a DTO for type-safe access
JMXMetricsDTO dto = Converters.standardConverter().convert(metrics)
                              .to(JMXMetricsDTO.class);

// Use metrics
System.out.println("System Load Average: " + dto.systemLoadAverage);
```

148.4 Framework Node Status Service

In case the cluster node is an OSGi framework, the FrameworkNodeStatus service is used to represent the node. FrameworkNodeStatus extends NodeStatus, and the node id is the OSGi framework UUID. Next to the Node Status service properties, this service has some additional service properties describing the OSGi and Java runtime:

Table 148.2 Additional service properties of the FrameworkNodeStatus service

Service Property Name	Type	Description
org.osgi.framework.version	String	The OSGi framework version.
org.osgi.framework.processor	String	The OSGi framework processor architecture.
org.osgi.framework.os_name	String	The OSGi framework operating system name.
java.version	String	The Java version.
java.runtime.version	String	The Java runtime version.
java.specification.version	String	The Java specification version.
java.vm.version	String	The Java VM version.

Similar to the Node Status service, the service properties of the Framework Node Status service can be converted to a FrameworkNodeStatusDTO to have type-safe access to these properties using the *Converter Specification* on page 1457.

The Framework Node Status service also extends the FrameworkManager interface, which provides a management interface for the OSGi framework. This allows a remote management agent to interact with the OSGi framework. The Framework Node Status service can be exported remotely with Remote Services, however alternative mechanisms to distribute this service are also permitted. For example, the FrameworkManager interface can also be made available through the *REST Management Service Specification* on page 717.

The following example uses the NodeStatus properties from a FrameworkNodeStatus service to see what country the framework is running in. If it is running in Germany a bundle specific for that country is installed:

```
@Component
public class FrameworkProvisioner {
    private static final Converter CONVERTER = Converters.standardConverter();
    @Reference(cardinality = MULTIPLE, policy = DYNAMIC)
    void addFramework(FrameworkNodeStatus fns, Map<String,Object> props) {
        // Convert the properties to the DTO for type safe access
        NodeStatusDTO dto = CONVERTER.convert(props).to(NodeStatusDTO.class);

        // Check the ISO 3166-1 alpha 3 country code
        if ("DEU".equals(dto.country)) {
            // If this framework runs in Germany, install a special bundle into it
            try {
                fns.installBundle("... Germany specific bundle ...");
            } catch (Exception e) {
                // log
            }
        }
    }
}
```

148.5 Application-specific Node Status metadata

The Node Status service provides a osgi.clusterinfo.tags property. Here, application specific tags can be assigned to the NodeStatus services. For example, one could assign different roles to the nodes such as "worker", "database", "storage", "gateway", etc. These roles are application-specific and should be defined by the application developer.

Bundles can specify additional tags to be included in the FrameworkNodeStatus service representing the current framework by registering any service with the service property org.osgi.service.clusterinfo.tags providing a custom String[] of tags. The Cluster Information implementation will add those to the tags property of the FrameworkNodeStatus service that represents the OSGi framework. For example:

```
// Register an arbitrary service that communicates the tags
// to be added to the osgi.clusterinfo.tags service property.
Dictionary<String, Object> props = new Hashtable<>();
props.put("org.osgi.service.clusterinfo.tags",
    new String [] {"database", "large_box"});
bundleContext.registerService(MyClass.class, this, props);
```

148.6 Security

148.6.1 Cluster Tag Permission

The ClusterTagPermission class allows fine-grained control over which bundles may add which tags to the Framework Node Status service. A bundle can be granted to add a certain tag to the Framework Node Status, or be granted to add any tag using the * wildcard.

148.6.2 Required Permissions

The Cluster Information Specification should only be implemented by trusted bundles. These bundles require ServicePermission[NodeStatus|FrameworkNodeStatus|FrameworkManager, REGISTER].

All bundles accessing the Cluster Information services should get ServicePermission[NodeStatus|FrameworkNodeStatus|FrameworkManager, GET].

Only trusted bundles who must be able to add Node Status tags should be assigned ClusterTagPermission[ClusterTag, ADD].

148.6.3 Remote service visibility in a cluster

By default, all remote OSGi services are visible within a cluster. This is handled by the *Remote Service Admin Service Specification* on page 493.

148.7 org.osgi.service.clusterinfo

ClusterInfo Services Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.clusterinfo; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.clusterinfo; version="[1.0,1.1)"

148.7.1 Summary

- ClusterTagPermission - A bundle's authority to add a tag to a NodeStatus service.
- FrameworkManager - Provides a management interface for accessing and managing a remote OSGi framework.
- FrameworkNodeStatus - The FrameworkNodeStatus service represents a node in the cluster that is also an OSGi framework.
- NodeStatus - The NodeStatus service represents a node in the cluster.

148.7.2 public final class ClusterTagPermission extends Permission

A bundle's authority to add a tag to a NodeStatus service.

148.7.2.1 public static final String ADD = "add"

The action string add.

148.7.2.2 public ClusterTagPermission(String tag, String actions)

tag Give permission to add this tag, use * wildcard to give permission to add any tag.

actions add.

□ Defines the authority to add a tag to the NodeStatus service.

148.7.2.3 public boolean equals(Object obj)

obj The object to test for equality with this ClusterTagPermission object.

□ Determines the equality of two ClusterTagPermission objects. This method checks that specified ClusterTagPermission has the same tag as this ClusterTagPermission object.

Returns true if obj is a ClusterTagPermission, and has the same tag as this ClusterTagPermission object; false otherwise.

148.7.2.4 public String getActions()

□ Returns the canonical string representation of the ClusterTagPermission action.

Always returns the ADD action.

Returns Canonical string representation of the ClusterTagPermission actions.

148.7.2.5 public int hashCode()

□ Returns the hash code value for this object.

Returns A hash code value for this object.

148.7.2.6 public boolean implies(Permission p)

p The target permission to interrogate.

□ Determines if the specified permission is implied by this object.

This method checks that the tag of the target is implied by the tag name of this object.

Returns true if the specified ClusterTagPermission action is implied by this object; false otherwise.

148.7.2.7 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing ClusterTagPermission objects.

Returns A new PermissionCollection object.

148.7.3 public interface FrameworkManager

Provides a management interface for accessing and managing a remote OSGi framework. This interface can be accessed remotely via Remote Services.

148.7.3.1 public BundleDTO getBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Retrieve the bundle representation for a given bundle Id.

Returns A BundleDTO for the requested bundle.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.2 public Map<String, String> getBundleHeaders(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the header for a bundle given by its bundle Id.

Returns Returns the map of headers entries.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.3 public Collection<BundleDTO> getBundles() throws Exception

□ Get the bundle representations for all bundles currently installed in the managed framework.

Returns Returns a collection of BundleDTO objects.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.4 public BundleStartLevelDTO getBundleStartLevel(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the start level for a bundle given by its bundle Id.

Returns Returns a BundleStartLevelDTO describing the current start level of the bundle.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.5 public int getBundleState(long id) throws Exception

id Addresses the bundle by its identifier.

□ Get the state for a given bundle Id.

Returns Returns the current bundle state as defined in org.osgi.framework.Bundle.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.6 public FrameworkStartLevelDTO getFrameworkStartLevel() throws Exception

□ Retrieves the current framework start level.

Returns Returns the current framework start level in the form of a FrameworkStartLevelDTO.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.7 public ServiceReferenceDTO getServiceReference(long id) throws Exception

id Addresses the service by its identifier.

□ Get the service representation for a service given by its service Id.

Returns The service representation as ServiceReferenceDTO.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.8 public Collection<ServiceReferenceDTO> getServiceReferences() throws Exception

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.9 public Collection<ServiceReferenceDTO> getServiceReferences(String filter) throws Exception

filter Passes a filter to restrict the result set.

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.10 public BundleDTO installBundle(String location) throws Exception

location Passes the location string to retrieve the bundle content from.

□ Install a new bundle given by an externally reachable location string, typically describing a URL.

Returns Returns the BundleDTO of the newly installed bundle.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.11 public void setBundleStartLevel(long id, int startLevel) throws Exception

id Addresses the bundle by its identifier.

startLevel The target start level.

□ Set the start level for a bundle given by its bundle Id.

Throws `Exception`– An exception representing a failure in the underlying remote call.

148.7.3.12 public void setFrameworkStartLevel(FrameworkStartLevelDTO startLevel) throws Exception

startLevel set the framework start level to this target.

□ Sets the current framework start level.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.13 public void startBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.14 public void startBundle(long id, int options) throws Exception

id Addresses the bundle by its identifier.

options Passes additional options as defined in org.osgi.framework.Bundle.start(int)

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.15 public void stopBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Stop a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.16 public void stopBundle(long id, int options) throws Exception

id Addresses the bundle by its identifier.

options Passes additional options as defined in org.osgi.framework.Bundle.stop(int)

□ Stop a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.17 public BundleDTO uninstallBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Uninstall a bundle given by its bundle Id.

Returns Returns the BundleDTO of the uninstalled bundle.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.18 public BundleDTO updateBundle(long id) throws Exception

id Addresses the bundle by its identifier.

□ Updates a bundle given by its bundle Id using the bundle-internal update location.

Returns Returns the BundleDTO of the updated bundle.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.3.19 public BundleDTO updateBundle(long id, String url) throws Exception

id Addresses the bundle by its identifier.

url The URL whose content is to be used to update the bundle.

□ Updates a bundle given by its URI path using the content at the specified URL.

Returns Returns the BundleDTO of the updated bundle.

Throws Exception– An exception representing a failure in the underlying remote call.

148.7.4 public interface FrameworkNodeStatus extends NodeStatus, FrameworkManager

The FrameworkNodeStatus service represents a node in the cluster that is also an OSGi framework.

148.7.5 public interface NodeStatus

The NodeStatus service represents a node in the cluster.

A node could represent an entity available in the network that is not necessarily running an OSGi framework, such as a database or a load balancer.

148.7.5.1 public Map<String, Object> getMetrics(String... names)

names a set of metric names that have to be obtained from the node. Of no names are specified all available metrics will be obtained. If a metric is requested that is not available by the node this metric is ignored and not present in the returned map.

□ Get the current metrics or other dynamic data from the node. Nodes may support custom metrics and as such the caller can request those metrics by name. The caller can specify the metric names to avoid having to compute and send all metrics over, if the caller is only interested in a subset of the available metrics.

Returns Map with the current node metrics

148.8 org.osgi.service.clusterinfo.dto

ClusterInfo DTO Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.clusterinfo.dto; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.clusterinfo.dto; version="[1.0,1.1)"
```

148.8.1 Summary

- `FrameworkNodeStatusDTO` - Data Transfer Object for a FrameworkNodeStatus Service.
- `NodeStatusDTO` - Data Transfer Object for a NodeStatus Service.

148.8.2 public class FrameworkNodeStatusDTO extends NodeStatusDTO

Data Transfer Object for a FrameworkNodeStatus Service.

148.8.2.1 public String java_runtime_version

The Java runtime version.

148.8.2.2 public String java_specification_version

The Java specification version.

148.8.2.3 public String java_version

The Java version.

148.8.2.4 public String java_vm_version

The Java VM version.

148.8.2.5 public String org_osgi_framework_os_name

The OSGi framework operating system name.

148.8.2.6 public String org_osgi_framework_processor

The OSGi framework processor architecture.

148.8.2.7 public String org_osgi_framework_version

The OSGi framework version.

148.8.2.8 public FrameworkNodeStatusDTO()

□ This DTO can be used to provide type safe access to properties of the FrameworkNodeStatus service.

148.8.3 public class NodeStatusDTO extends DTO

Data Transfer Object for a NodeStatus Service.

148.8.3.1 public String cluster

The name of the cluster this node belongs to.

148.8.3.2 public String country

ISO 3166-1 alpha-3 location where this node instance is running, if known.

148.8.3.3 public String[] endpoints

The endpoint(s) at which this node can be accessed from the viewpoint of the consumer of the service.

148.8.3.4 public String id

The globally unique ID for this node. For example the Docker ID if this node is a Docker container, or the framework UUID if this node is an OSGi framework.

148.8.3.5 public String location

ISO 3166-2 location where this node instance is running, if known. This location is more detailed than the country code as it may contain province or territory.

148.8.3.6 public String parentid

An optional parentID indicating this node is part of or embedded in another node. For example multiple virtual machines could run on the same physical node.

148.8.3.7 public static final String PREFIX_ = "osgi.clusterinfo."

Prefix used for the converter

148.8.3.8 public String[] privateEndpoints

Private endpoint(s) at which this node can be accessed from within the cluster only.

148.8.3.9 public String region

Something smaller than a country and bigger than a location (e.g. us-east-1 or other cloud-specific location)

148.8.3.10 public String[] tags

Tags associated with this node that can be contributed to by the provider and also by bundles.

148.8.3.11 public String vendor

The vendor name of the cloud/environment in which the node operates.

148.8.3.12 public String version

The version of the cloud/environment in which the node operates. The value follows the versioning scheme of the cloud provider and may therefore not comply with the OSGi versioning syntax.

148.8.3.13 public String zone

Regions are often subdivided in zones that represent different physical locations. The zone can be provided here.

148.8.3.14 public NodeStatusDTO()

□ This DTO can be used to provide type safe access to properties of the NodeStatus service.

149 Device Service Specification for ZigBee™ Technology

Version 1.0

149.1 Introduction

The [1] *ZigBee Specification* is a standard wireless communication protocol designed for low-cost and low-power devices from the ZigBee Alliance. ZigBee is widely supported by various types of devices such as smart meters, lights and many kinds of sensors in the residential area. OSGi applications need to communicate with those ZigBee devices.

This specification defines how OSGi bundles can be developed to discover and control ZigBee devices on the one hand, and act as ZigBee devices and interoperate with ZigBee clients on the other hand. In particular, a Java mapping is provided for the standard hierarchical representation of ZigBee devices called the ZigBee Cluster Library. The [2] *ZigBee Cluster Library Specification* also describes the external API of a ZigBee Base Driver based upon the OSGi Device Access Specification.

149.2 Essentials

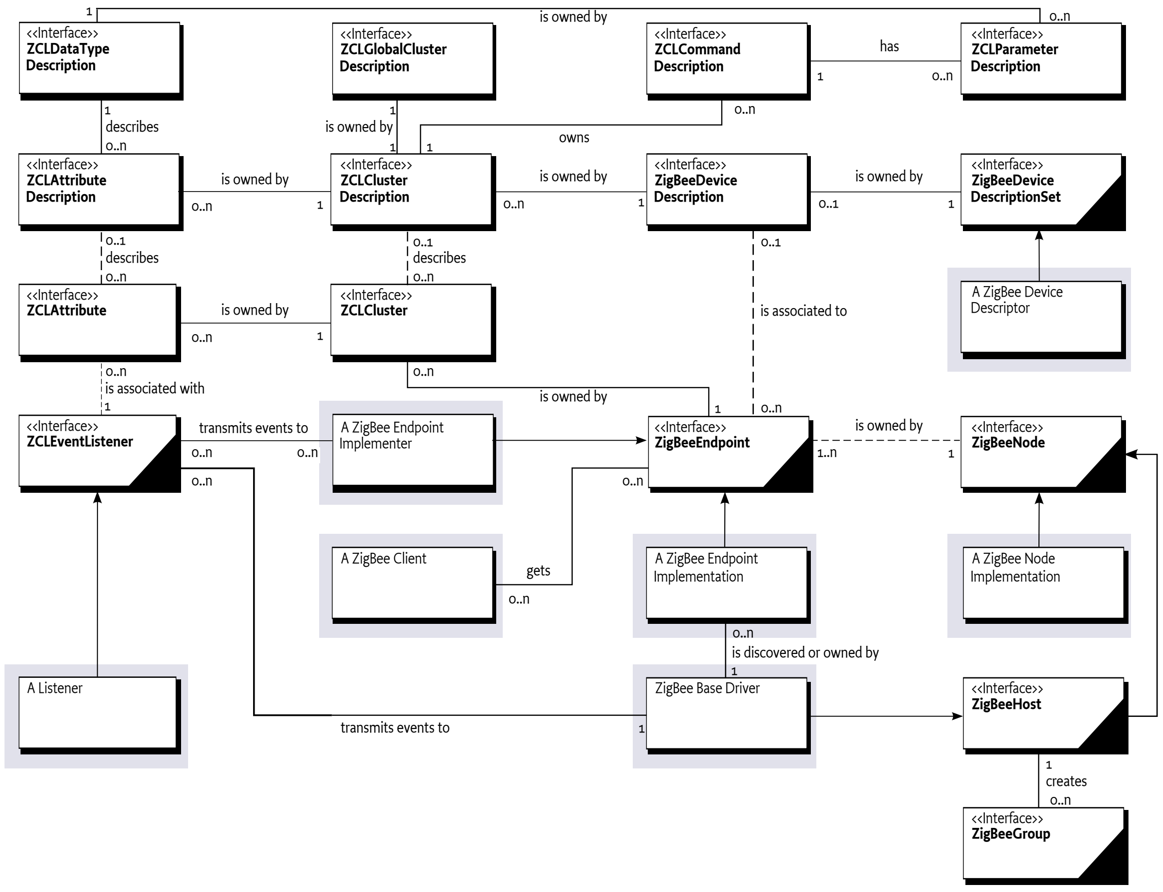
- *Scope* – This specification is limited to general device discovery and control aspects of the ZigBee and the ZigBee Cluster Library specifications. Aspects concerning the representation of specific ZigBee profiles are not addressed.
- *Transparency* – ZigBee devices discovered on the network and devices locally implemented on the platform are represented in the OSGi service registry with the same API.
- *Lightweight implementation option* – The full description of ZigBee device services on the OSGi platform is optional. Some base driver implementations may implement all the classes including ZigBee device description classes while implementations targeting constrained devices are able to implement only the part that is necessary for ZigBee device discovery and control.
- *Network Selection* – It must be possible to restrict the use of the ZigBee protocols to a selection of the connected networks.
- *Logical node type selection* – It is possible to make an OSGi-based device appearing as a ZigBee end device, a ZigBee router or a ZigBee coordinator.
- *Event handling* – Bundles are able to listen to ZigBee events.
- *Discover and Control ZigBee Endpoints as OSGi services* – available ZigBee endpoints are dynamically reified as OSGi services in the service registry.
- *Export OSGi services as ZigBee Endpoints* – available ZigBee endpoints are dynamically reified as OSGi services in the service registry.

149.3 Entities

- *ZigBee Base Driver* – The bundle that implements the bridge between OSGi and ZigBee networks.

- *ZigBee Node* – A physical ZigBee node. This entity is represented by a ZigBeeNode object. It is registered as an OSGi service by the Base Driver.
- *ZigBee Endpoint* – A logical device that defines a communication entity within a ZigBee node through which a specific application profile is carried. This concept is represented by a ZigBeeEndpoint object. Registered as an OSGi service, an endpoint can be local (implemented on the Framework) or external (implemented by another device on the network).
- *ZigBee Device Description* – Statically describes a ZigBee endpoint by providing its input/output clusters and specifies which of described commands and attributes are mandatory or not. This entity is represented by a ZigBeeDeviceDescription object.
- *ZigBee Device Description Set* – A service representing a set of ZigBeeDeviceDescription objects.
- *ZigBee Cluster* – Represents a ZigBee cluster entity, that is, a set of attributes and commands. It allows the read and write of attribute values, and allows command invocation. This concept is represented by a ZCLCluster object.
- *ZigBee Cluster Description* – Cluster description provides details about available commands and attributes for a specific Cluster. A cluster description should be constant. A cluster description holds either a Client or a Server Cluster description and refers to a global cluster description.
- *ZigBee Global Cluster Description* – Global cluster description holds the server and client cluster description as well as common information such as cluster id, description and name. This concept is represented by a ZCLGlobalClusterDescription object.
- *ZigBee Command Description* – Statically describes a specific cluster command by giving its name, id, parameters. This entity is represented by a ZCLCommandDescription object.
- *ZigBee Parameter Description* – A ZigBee parameter description has a name, a range and a data type. This entity description is represented by a ZCLParameterDescription object.
- *ZigBee Attribute* – Holds the current value of an existing cluster attribute, it allows easy (de)encoding. This concept is represented by a ZCLAttribute object.
- *ZigBee Attribute Description* – Statically describes a ZigBee Attributes (data type, name, default value). It does not hold any current value. This concept is represented by a ZCLAttributeDescription object.
- *ZigBee Event Listener Service* – A service that listens to events coming from ZigBee devices.
- *ZigBee Event* – An event generated by a ZigBee node. It contains a modified attribute value of a specific cluster. This concept is represented by a ZigBeeEvent object.
- *ZigBee Command Response Stream* – A stream is a helper that manages asynchronous responses from several endpoints that received a same request message. This entity is represented by a ZigBeeCommandResponseStream. For methods that generates a message to a unique endpoint, a Promise is used instead.
- *ZigBee Host* – The machine that hosts the code to run a ZigBee device or client. It contains information related to the Host. If the host is in the coordinator logical node type, it enables networking configuration. It is registered as an OSGi service. This concept is represented by ZigBeeHost.
- *ZigBee Client* – An application that is intended to control ZigBee devices services.
- *ZigBee Group* – Enables group management. It is registered as an OSGi service.

Figure 149.1 ZigBee Service Specification class Diagram org.osgi.service.zigbee package



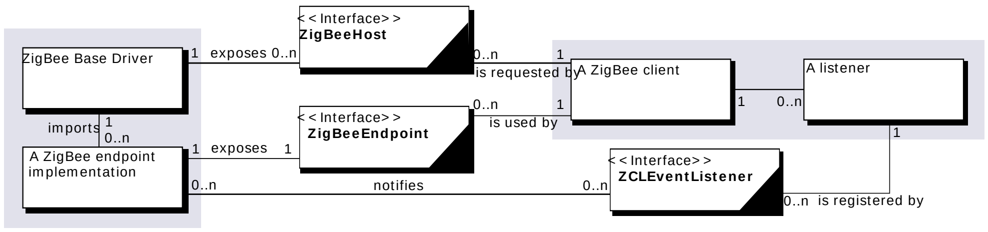
<<Interface>> ZCLDataType Description
<<Interface>> ZCLGlobalCluster Description
<<Interface>> ZCLCommand Description
<<Interface>> ZCLParameter Description
<<Interface>> ZCLAttribute Description
<<Interface>> ZCLCluster Description
<<Interface>> ZigBeeDevice Description
<<Interface>> ZigBeeDevice DescriptionSet
<<Interface>> ZCLAttribute
<<Interface>> ZCLCluster
A ZigBee Device Descriptor
<<Interface>> ZCLEventListener
A ZigBee Endpoint Implementer
<<Interface>> ZigBeeEndpoint
<<Interface>> ZigBeeNode
A ZigBee Client
A ZigBee Endpoint Implementation
A ZigBee Node Implementation
A Listener
ZigBee Base Driver
<<Interface>> ZigBeeHost
<<Interface>> ZigBeeGroup
is owned by
has
describes
owns
is associated to
is associated with
transmits events to
gets
is discovered or owned by
creates
1
0..n
0..1
1..n

149.4 Operation Summary

OSGi applications interact with ZigBee devices through their object representation (proxies) registered in OSGi service registry. To make a ZigBee device available as an OSGi service to ZigBee clients on the framework, an OSGi service object must be registered under the ZigBeeNode interface with the OSGi framework and an OSGi service must be registered under the ZigBeeEndpoint interface with the OSGi framework for every endpoint that is contained by the ZigBee node.

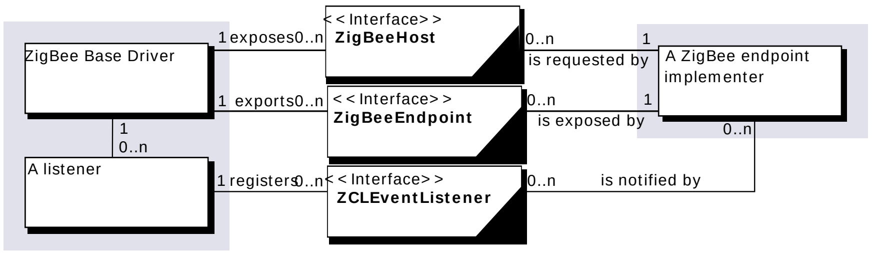
The ZigBee Base Driver is responsible for mapping networked devices into ZigBeeNode and ZigBeeEndpoint objects, through the use of a ZigBee radio chip. The latter is represented on the OSGi framework as an object implementing ZigBeeHost interface. This is called a *device import* situation (see Figure 149.2 on page 1020).

Figure 149.2 *ZigBee device import*



OSGi bundles may also expose framework-internal (local) ZigBeeEndpoint instances, registered within the framework (see Figure 149.3 on page 1020). The Base Driver then should emulate those objects as ZigBee endpoints associated to the ZigBee node represented by the underlying ZigBee host (ZigBee chip) on the ZigBee network. This is a *device export* situation. For more information about this process, please report to section *Implementing a ZigBee Endpoint* on page 1030.

Figure 149.3 *ZigBee device export*



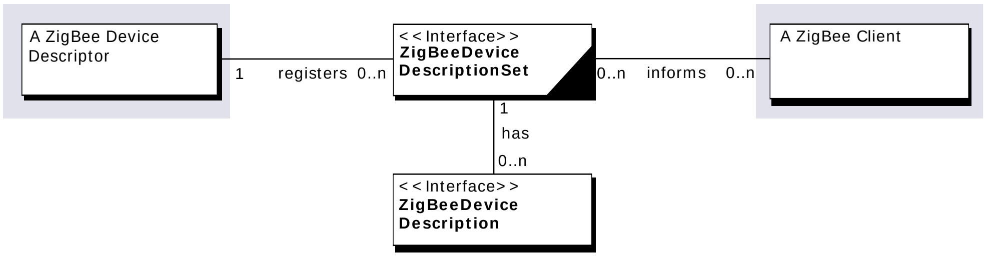
To control ZigBee devices, a bundle should track ZigBeeEndpoint services in the OSGi service registry and control them appropriately. OSGi applications can browse the clusters (ZCLCluster objects) that are discovered on every registered ZigBeeEndpoint and attributes (ZCLAttribute objects) that are discovered on every ZCLCluster. They can invoke commands on these clusters and get the current value of attributes.

Several methods obey an asynchronous mechanism. For instance, ZigBee command invocation is made through the call to ZCLCluster invoke method that returns a Promise. When the command response is received, the Promise is resolved and Promise.getValue() returns the expected response value. The Promise is resolved by the base driver in the device import situation and by the invoked local ZCLCluster in the device export situation. A ZCLCommandResponseStream is used instead of a Promise in case of a method that generates a message broadcast (or groupcast) to potentially several endpoints.

OSGi bundles – called listeners in Figure 149.1 – subscribe to attribute value changes through the Whiteboard Pattern ([6] *Listeners considered harmful: The whiteboard pattern*). They register an object under the ZCLEventListener interface with properties identifying a ZigBee attribute and a special event filter. This registration is conveyed as a ZigBee configure report command on the ZigBee network in the device import situation. Reports are received by the base driver and transmitted as notifyEvent(ZigBeeEvent) method calls on relevant ZCLEventListener services in this situation. Local ZigBeeEndpoint objects directly call these methods to notify listeners with reports in the export situation. The Base Driver conveys events received through a ZCLEventListener to networked the ZigBee endpoints that have subscribed to relevant reports.

Endpoints, clusters, commands and attributes are specified by ZigBee Alliance or vendor-specific descriptions. Those descriptions may be provided on the OSGi platform by any bundle through the registration of ZigBeeDeviceDescriptionSet services (see Figure 149.4 on page 1021). Every service is a set of descriptions that enables applications to retrieve information about the clusters, commands, attributes supported by the described type of endpoint.

Figure 149.4 *Using a set of device descriptions*

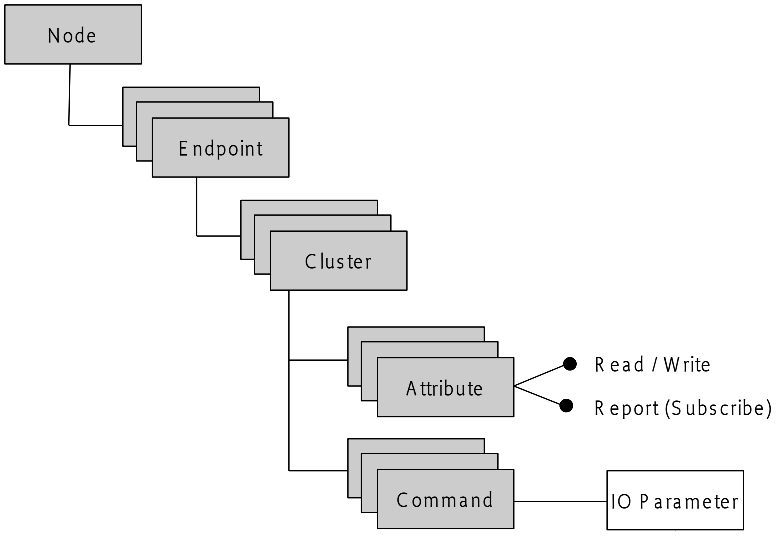

A ZigBee Device Descriptor
1
registers
0..n
<<Interface>> ZigBeeDevice DescriptionSet
0..n
informs
0..n
A ZigBee Client
1
has
0..n
<<Interface>> ZigBeeDevice Description

149.5 ZigBee Base Driver

Most of the functionality described in the operation summary is implemented in a ZigBee base driver. A ZigBee base driver is a bundle that implements the ZigBee protocols and handles the interaction with bundles that use the ZigBee devices. It must discover ZigBee devices on the ZigBee network and map each discovered device into an OSGi registered ZigBeeNode service. It must also export, on the ZigBee Network, ZigBeeEndpoint services (programmatically registered as OSGi services).

ZigBeeNode object also provides simple methods to handle standard ZigBee Device Object networking features: getLinksQuality(), getRoutingTable(), and leave().

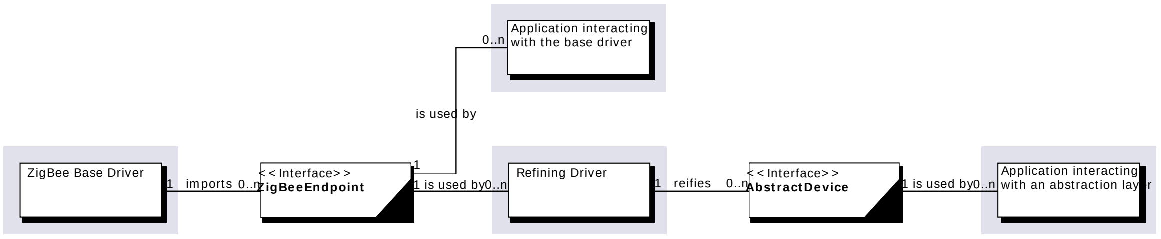
Figure 149.5 *ZigBee Cluster Library model*



All interfaces corresponding to the ZigBee Cluster Library model (see Figure 149.5 on page 1021) must be implemented in order to discover and control asynchronously ZigBee devices. Classes related to the description of these entities named with suffix Description may optionally be implemented. This rule follows the fact that ZigBee device descriptions are not downloadable on the device itself and are often given to developers in an out-of-band manner.

Several base drivers may be deployed on a residential OSGi device, one for every supported network technology. An OSGi device abstraction layer may then be implemented as a layer of refining drivers above a layer of base drivers. The refining driver is responsible for adapting technology-specific device services registered by the base driver into device services of another model (see AbstractDevice interface in Figure 149.6 on page 1022). In the case of a generic device abstraction layer, the model is agnostic to technologies.

Figure 149.6 *The ZigBee Base Driver and a refining driver representing devices in an abstract model*


Application interacting with the base driver
0..n
is used by
1
ZigBee Base Driver
1 imports 0..n
<<Interface>> ZigBeeEndpoint
1 is used by 0..n
Refining Driver
1 reifies 0..n
<<Interface>> AbstractDevice
1 is used by 0..n
Application interacting with an abstraction layer

The ZigBee Alliance defines their own abstract model with ZigBee Profiles, for example, Home Automation, Lighting, and refining drivers may provide the implementation of all ZigBee standard devices with ZigBee-specific Java interfaces. The AbstractDevice interface of Figure 149.6 on page 1022 is then replaced by a ZigBee-specific Java interface in that case. The need and the choice of the abstraction depends on the targeted application domain.

149.6 ZigBee Node

A ZigBee node represents a physical ZigBee device and should adhere to a specific application profile that can be either public or private. Profiles define the environment of the application, the type of devices and the clusters used for them to communicate.

A physical device is reified and registered as a ZigBeeNode service in the Framework. A ZigBee node holds several ZigBee endpoints that are registered as ZigBeeEndpoint objects.

ZigBee nodes properties are defined in the ZigBee Specification. These properties must be registered in the OSGi Framework services registry so they are searchable. ZigBeeNode must be registered with the following properties:

- IEEE_ADDRESS – (zigbee.node.ieee.address/BigInteger) specifies the IEEE Address of a ZigBee node.
- LOGICAL_TYPE – (zigbee.node.description.node.type/Short) specifies a device logical type.
- MANUFACTURER_CODE – (zigbee.node.description.manufacturer.code/Integer) specifies a manufacturer code that is allocated by the ZigBee Alliance, relating to the device manufacturer.
- POWER_SOURCE – (zigbee.node.power.source/Boolean) is the ZigBee power source, that is, 3rd bit of "MAC Capabilities" in Node Descriptor, which is set to 1 if the current power source is mains power, set to 0 otherwise.
- RECEIVER_ON_WHEN_IDLE – (zigbee.node.receiver.on.when.idle/Boolean) represents the ZigBee receiver on when idle, that is, 4th bit of "MAC Capabilities" in Node Descriptor, which is set to 1 if the device does not disable its receiver to conserve power during idle periods, set to 0 otherwise.
- PAN_ID – (zigbee.node.pan.id/Integer) (Personal Area Network Identifier) is a 16-bit value that identifies a ZigBee network. Every ZigBeeNode object is associated to a PAN ID, which can be retrieved through the getPanId() method.
- EXTENDED_PAN_ID – (zigbee.node.pan.extended.id/BigInteger) Extended PAN ID is a 64-bit numbers that uniquely identify a PAN. It is intended to enhance selection of a PAN and enable recognition of network after PAN ID change (due to a previous conflict). getExtendedPanId() returns the network extended PAN ID if specified.

  Note: PAN_ID and EXTENDED_PAN_ID are optional, but at least one of these properties MUST be specified.
- DEVICE_CATEGORY (see the OSGi Device Access Specification) – (DEVICE_CATEGORY) describes a table of the categories to which the device belongs. One of the values MUST be “ZigBee” (DEVICE_CATEGORY).

Additional properties (defined in the OSGi Device Access Specification) may be set:

- DEVICE_DESCRIPTION – if the complex descriptor of the device is available, the value MUST be set and MUST be the value returned by getModelName().
- DEVICE_SERIAL – if the complex descriptor of the device is available, the value MUST be set and MUST be the value returned by getSerialNumber().

Finally, service.pid property MUST be set.

ZigBee nodes describes themselves using descriptor data structures:

- getNodeDescriptor() – Returns a Promise object that is asynchronously resolved with a ZigBeeNodeDescriptor object representing the Node Descriptor which contains information about the node capabilities. On failure, the promise is resolved with an exception instead.
- getComplexDescriptor() – Returns a Promise object that is asynchronously resolved with a ZigBeeComplexDescriptor object representing the Complex Descriptor which contains extended information for each device description contained in this node. On failure, the promise is resolved with an exception instead, especially an exception with NO_DESCRIPTOR error code if no Complex Descriptor is provided.
- getPowerDescriptor() – Returns a Promise object that is asynchronously resolved with a ZigBeePowerDescriptor object representing the Power Descriptor which contains power-related information of this node. On failure, the promise is resolved with an exception instead, especially an exception with NO_DESCRIPTOR error code if no Power Descriptor is provided.
- getUserDescription() – Returns a Promise object that is asynchronously resolved with the unique field named "User description" of the User Descriptor, which contains information that allows the user to identify the device using user-friendly character string. On failure, the promise is resolved with an exception instead, especially an exception with NO_DESCRIPTOR error code if no User Descriptor is provided.

ZigBeeNode objects provide invoke methods to send network frames within ZDP layer, while invoking ZigBee Cluster Library (ZCL) commands is enabled on ZCLCluster objects. ZCL commands can be however broadcast on a ZigBee node thanks to broadcast methods. Broadcasting enables the sending of a ZCL command to all clusters identified with an identifier of all endpoints available on the targeted ZigBee node.

All discovered ZigBee nodes in the local networks are registered under the ZigBeeNode interface within the OSGi Framework. Every time a ZigBee node appears or quits the network, the associated OSGi service is registered or unregistered in the OSGi service registry. Thanks to the ZigBee Base Driver, the OSGi service availability in the registry mirrors ZigBee device availability on ZigBee networks. Using a remote ZigBee node thus involves tracking ZigBeeNode services in the OSGi service registry. The following code illustrates how this can be done. The sample Controller class extends the ServiceTracker class so that it can track all ZigBeeNode services and add them to a user interface, such as a remote controller application. The friendly name of this node is retrieved in order to be printed on the user interface.

```
class Controller extends ServiceTracker {
    UI ui;
    Controller( BundleContext context ) {
        super( context, ZigBeeNode.class, null );
    }
    public Object addingService( ServiceReference ref ) {
        ZigBeeNode node = (ZigBeeNode)super.addingService(ref);
        ui.addNode( node );
        return node;
    }
    public void removedService( ServiceReference ref, Object endpoint ) {
        ui.removeNode( (ZigBeeNode) node );
        super.removedService(ref);
    }
    ...
}

public class UI {
    public void addNode(ZigBeeNode node) {
        final Promise p = node.getUserDescription();
        p.onResolve(new Runnable() {
            public void run() {
                try {
                    String friendlyName = (String) p.getValue();
                    createUINode(node, friendlyName);
                } catch (InvocationTargetException e) {
                     log.info("Get User Description command returned "
                               + "a failure: " + e.getCause() + ".");
                     createUINode(node, "No friendly name");
                } catch (InterruptedException e) {
                     e.printStackTrace();
                }
            }
        });
    }
...
}
```

149.7 ZigBee Endpoint

Communication between devices is done through an addressable component called ZigBee endpoint which holds a number of ZigBee clusters. A ZigBee cluster represents a functional unit in a device.

An endpoint defines a communication entity within a device through which a specific application is carried. So, it represents a logical device object used for communication.

For example, a remote control light might allocate Endpoint 7 for the control of lights in the master bedroom, Endpoint 9 to manage the heating and air conditioning system, and Endpoint 14 for controlling the security system.

The ZigBee specification defines that a maximum of 240 Endpoints is allowed per ZigBeeNode. Endpoint 0, also called the ZigBee Device Object (ZDO), is reserved for the management operations on both ZigBee node and ZigBee endpoints, endpoint 255 is reserved for broadcasting to all endpoints, endpoints 241-254 are reserved for future use.

Endpoint 0 and endpoint 255 capabilities are not exposed, only endpoints 1-240 should be registered as services. Endpoints are registered under the ZigBeeEndpoint interface with the following properties:

- IEEE_ADDRESS – (zigbee.node.ieee.address/BigInteger) specifies the IEEE Address of the parent node.
- ENDPOINT_ID – (zigbee.endpoint.id/Short) specifies the endpoint address within the node. Applications shall only use endpoints 1-240.
- PROFILE_ID – (zigbee.device.profile.id/Integer) identifies the profile that the endpoint belongs to. The profile can be either a ZigBee Alliance standard profile or a vendor-specific profile. The ZigBee specification defines several profile identifiers, and some others are vendor specific.

- HOST_PID – (zigbee.endpoint.host.pid/String) – The ZigBee local host identifier is intended to uniquely identify the ZigBee local host, since there could be many hosts on the same platform. All the endpoints that belong to a specific network MUST specify the value of the associated host number.
- DEVICE_ID – (zigbee.device.id/Integer) identifies the device description supported by this endpoint. Like for profile identifiers, the ZigBee specification defines several device identifiers. Vendors are also able to define specific device identifiers.
- DEVICE_VERSION – (zigbee.device.version/Integer) specifies the device description version supported by this endpoint.
- INPUT_CLUSTERS – (zigbee.endpoint.clusters.input/Integer[]) specifies the list of input cluster ids supported by this endpoint. Input cluster are called Server cluster.
- OUTPUT_CLUSTERS – (zigbee.endpoint.clusters.output/Integer[]) specifies the list of output cluster ids supported by this endpoint. Output cluster are called Client cluster.
- DEVICE_CATEGORY (see the OSGi Device Access Specification) – (DEVICE_CATEGORY) describes a table of the categories to which the device belongs. One of the values MUST be “ZigBee” (DEVICE_CATEGORY).

Finally, service.pid property MUST be set. In device import case, it is a free unique identifier that enables OSGi ZigBee clients to identify any imported endpoint across node reboots. It may concatenate the endpoint IEEE address, a separator, for example, '_', and the endpoint ID. In endpoint export case, it is a free unique identifier that enables the base driver to identify any exported endpoint across local bundle restarts. In this case, service.pid property may concatenate bundle identifier, a separator, for example, '_', and a number.

A ZigBeeEndpoint may contain a number of input or output clusters. ZigBeeEndpoint provides getServerCluster(int) and getClientCluster(int) to return a specific server input or client output cluster.

Every endpoint must provide a simple descriptor. getSimpleDescriptor() returns a Promise object that is asynchronously resolved with a ZigBeeSimpleDescriptor object which contains general information about the endpoint or with an exception in case of a failure.

ZigBeeEndpoint interface provides two methods to bind and unbind ZigBee clusters: bind(String,int) and unbind(String,int). The entity that wants to bind clusters is responsible for initializing, maintaining and removing the bindings across ZigBeeEndpoint service events. This entity is the local OSGi Application that asked this binding or the ZigBee Base Driver if the binding has been requested by a remote ZigBee node.

ZigBeeEndpoint interface provides a getBoundEndPoints(int) method that provides the table of bound ZigBeeEndpoint objects identified by their service PIDs.

149.8 ZigBee Device Description

A ZigBee endpoint may have a description used to describe his input and output clusters, and which of these clusters are mandatory or optional. A ZigBeeDeviceDescription object provides associated information about an endpoint.

149.9 ZigBee Device Description Set

ZigBeeDeviceDescriptionSet objects may be registered as OSGi services by any bundle. A ZigBeeDeviceDescriptionSet provides getDeviceSpecification(int,short) which returns the device description, if provided, for the identified endpoint, or null otherwise. A ZigBeeDeviceDescriptionSet service should be registered with the following properties:

- VERSION – (zigbee.profile.version/Short) The application profile version.
- PROFILE_ID – see ZigBeeEndpoint.PROFILE_ID property.
- PROFILE_NAME – (zigbee.profile.name/String) The profile name.
- MANUFACTURER_CODE – see ZigBeeNode.MANUFACTURER_CODE property.
- DEVICES – (zigbee.profile.devices/Integer[]) comma separated list of devices identifiers supported by the set.

149.10 ZCL Cluster

Devices communicate with each other by means of clusters, which may be inputs to or outputs of the device. For example, ZigBee Home Automation profile provides a cluster dedicated to the control of lighting subsystems. Clusters are represented under ZCLCluster interface.

ZCLCluster objects combine one or more ZigBee commands (or frames) and ZCLAttribute objects.

ZCLCluster provides some methods for reading and writing attributes values:

- readAttributes(ZCLAttributeInfo[]) – The ZigBee Base driver MAY generate the read attributes command on behalf of the OSGi application that is invoking this method. The latter returns a Promise object that is asynchronously resolved with a Map of ZCLReadStatusRecord identified by their attribute identifiers. On failure, the promise is resolved with an exception instead.
- writeAttributes(boolean,Map) – The ZigBee Base driver generates the write attributes command on behalf of the OSGi application that is invoking this method. The boolean undivided parameter specifies that if any attribute cannot be written, for example, if an attribute is not implemented on the device, or a value to be written is outside the valid range, no attribute values are changed.

ZCLCluster objects use ZCLFrame to invoke ZigBee commands :

- invoke(ZCLFrame) – a sequence of bytes represents the command frame. The source endpoint is not specified in this method call. To send the appropriate message on the network, the base driver must generate a source endpoint. The latter must not correspond to any exported endpoint.
- invoke(ZCLFrame,String) – a sequence of bytes represents the command frame, and exportedServicePID is the source endpoint of the command request. In targeted situations, the source endpoint is the valid service PID of an exported endpoint.

  A Promise is returned and manages the command response asynchronously.

149.11 ZCL Cluster Description

A ZCLClusterDescription describes the server or client part of a ZCLCluster. It lists the available commands and attributes for this client or server cluster.

Every cluster client and server may have attributes (see [2] *ZigBee Cluster Library Specification*, Chapter 2.2.1), received and generated commands. ZCLClusterDescription provides methods to describe commands, attributes and retrieve general cluster information.

149.12 ZCL Global Cluster Description

ZCLGlobalClusterDescription describes a cluster general information: id, name, description. It provides the ZCLClusterDescription for both client and server part of this cluster.

149.13 ZigBee Command Description

ZCLCommandDescription describes a ZigBee command.

ZCLCommandDescription contains ZCLParameterDescription objects which describe the command parameters.

All clusters (server and client) shall support generation, reception and execution of the default response command.

Every cluster (server or client) that implements attributes shall support reception of, execution of, and response to all commands to discover, read, write, report, configure reporting of, and read reporting configuration of these attributes. Generation of these commands is application dependent.

149.14 ZigBee Attribute

A ZigBee cluster is associated with a set of attributes. Every attribute is represented by an object implementing ZCLAttribute interface extending ZCLAttributeInfo. ZCLAttribute provides getValue() and setValue(Object) to retrieve and set the current attribute value with the use of a Promise, which returns the response asynchronously.

149.15 ZigBee Attribute Description

A ZCLAttributeDescription also extends ZCLAttributeInfo and describes information about a specific ZCLAttribute.

149.16 ZCL Data Type Description

ZCLAttributeInfo and ZCLParameterDescription provide getDataType() and getDataTypeDescription() methods, respectively, which return ZCLDataTypeDescription objects. One object is associated to every ZigBee data type, see ZigBeeDataTypes constants in ZigBee Data Types section below.

149.17 ZCL Simple Type Description

ZCLSimpleTypeDescription extends ZCLDataTypeDescription interface to provide the following methods:

- serialize(ZigBeeDataOutput,Object) – Serializes a Java object corresponding to the Java data type given by getJavaDataType(), and adds the result to the given ZigBeeDataOutput according to ZigBee Cluster Library.
- deserialize(ZigBeeDataInput) – Deserializes the given data into a Java object of the Java data type given by getJavaDataType().

Every ZigBee data type is associated to a ZCLSimpleTypeDescription implementation, except ZigBee Array, Bag, Set, and Structure types.

149.18 Promise and Response Stream objects

Promise and ZCLCommandResponseStream objects handle ZigBee network communication latency and errors. An org.osgi.util.promise.Promise is immediately returned by every method that generates a message exchange with one ZigBee endpoint, that is, the sending of a message to this endpoint and the handling of a unique response from this endpoint. No exception is thrown by this method. The Promise handles the expected result and any occurring error asynchronously. The caller can either get a callback when the Promise is resolved with a value or an error, or the Promise can be used in chaining. Both onResolve(Runnable) callbacks and then(Success, Failure) chaining can be repeated any number of times, even after the Promise has been resolved. When the Promise is resolved, callbacks and chaining are called, Promise.isDone() returns true, and either Promise.getValue() returns a value or Promise.getFailure() returns a relevant Throwable. The type of the value and the type of Throwable are specific to the method returning the Promise. In import situations, the base driver fails the Promise when a timeout is reached before any response is received on the network. The returned failure is then a ZigBeeException with TIMEOUT error code. The associated timeout is given by getCommunicationTimeout(). It can be set by calling setCommunicationTimeout(long) on the appropriate ZigBeeHost object.

A ZCLCommandResponseStream is immediately returned by every method that generates the sending of a message to potentially several endpoints with the expectation of a response from several of them. No exception is thrown by this method. The caller can register a handler with forEach(Predicate). The unique method of the handler is called with a ZCLCommandResponse every time a response is received from one of the targeted endpoints until the ZCLCommandResponseStream is closed. The latter is closed either when the handler returns false to the test method or close() is called. ZCLCommandResponseStream is used for the following message invocation types:

- Broadcasting: Sending a message to all available endpoints of a specific type and receiving responses from each of them, see broadcast(int,ZCLFrame,String).
- Groupcasting: Sending a message to the endpoints of a specific type in a group of endpoints and receiving responses from each of them, see groupcast(int,ZCLFrame,String).
- Nodecasting: Sending a message to the endpoints of a specific type on a node and receiving responses from each of them, see broadcast(int,ZCLFrame,String).

149.19 ZigBee Data Types

The ZigBeeDataTypes class provides all standard ZigBee data type identifiers as constants. It should be noted that actual value of these constants do not necessarily match the values assigned in the ZCL specification for these data types.

The org.osgi.service.zigbee.types package contains an implementation class for each of the ZCL scalar data types, with the exception of NO_DATA and UNKNOWN . Each of these classes declares a static getInstance() method, that returns a singleton of the class itself. Moreover, because they implement the ZCLSimpleTypeDescription interface, they provide methods for getting some metadata information about the ZCL data type they represent, like the relative ZigBeeDataTypes constant (getId() method) and the Java class the ZCL data type is mapped to. Methods to marshal and unmarshal the data type into a ZigBeeDataInput stream and from a ZigBeeDataOutput stream according to the ZigBee specification, are provided as well.

Here is the table of encoding relations between ZigBee types and Java types, used in this specification:

Table 149.1 Mapping of ZCL Data Types to Java

ZigBeeDataType constant	ZigBee type	Java Type
NO_DATA	No data	
GENERAL_DATA_8	8-bit data	Byte
GENERAL_DATA_16	16-bit data	Short
GENERAL_DATA_24	24-bit data	Integer
GENERAL_DATA_32	32-bit data	Integer
GENERAL_DATA_40	40-bit data	Long
GENERAL_DATA_48	48-bit data	Long
GENERAL_DATA_56	56-bit data	Long
GENERAL_DATA_64	64-bit data	Long
BOOLEAN	Boolean	Boolean
BITMAP_8	8-bit bitmap	Byte
BITMAP_16	16-bit bitmap	Short
BITMAP_24	24-bit bitmap	Integer
BITMAP_32	32-bit bitmap	Integer
BITMAP_40	40-bit bitmap	Long
BITMAP_48	48-bit bitmap	Long
BITMAP_56	56-bit bitmap	Long
BITMAP_64	64-bit bitmap	Long
UNSIGNED_INTEGER_8	Unsigned 8-bit integer	Short
UNSIGNED_INTEGER_16	Unsigned 16-bit integer	Integer
UNSIGNED_INTEGER_24	Unsigned 24-bit integer	Integer
UNSIGNED_INTEGER_32	Unsigned 32-bit integer	Long
UNSIGNED_INTEGER_40	Unsigned 40-bit integer	Long
UNSIGNED_INTEGER_48	Unsigned 48-bit integer	Long
UNSIGNED_INTEGER_56	Unsigned 56-bit integer	Long
UNSIGNED_INTEGER_64	Unsigned 64-bit integer	BigInteger
SIGNED_INTEGER_8	Signed 8-bit integer	Byte
SIGNED_INTEGER_16	Signed 16-bit integer	Short
SIGNED_INTEGER_24	Signed 24-bit integer	Integer
SIGNED_INTEGER_32	Signed 32-bit integer	Integer
SIGNED_INTEGER_40	Signed 40-bit integer	Long
SIGNED_INTEGER_48	Signed 48-bit integer	Long
SIGNED_INTEGER_56	Signed 56-bit integer	Long
SIGNED_INTEGER_64	Signed 64-bit integer	Long
ENUMERATION_8	8-bit enumeration	Short
ENUMERATION_16	16-bit enumeration	Integer
FLOATING_SEMI	Semi-precision float	Float
FLOATING_SINGLE	Single precision float	Float
FLOATING_DOUBLE	Double	Double
CHARACTER_STRING	Character string	String
OCTET_STRING	Octet string	byte[]
LONG_CHARACTER_STRING	Character string	String
LONG_OCTET_STRING	Octet string	byte[]
ARRAY	Array	

ZigBeeDataType constant	ZigBee type	Java Type
STRUCTURE	Structure	
SET	Set	
BAG	Bag	
CLUSTER_ID	Cluster ID	Integer
ATTRIBUTE_ID	Attribute ID	Integer
BACNET_OID	BACnet OID[1] (Unsigned 32-bit integer)	Long
TIME_OF_DAY	Time of day	byte[4]
DATE	Date	byte[4]
UTC_TIME	UTC Time	Long
IEEE_ADDRESS	IEEE address (MAC-48,EUI-48/64)	BigInteger
SECURITY_KEY_128	128-bit Security Key	byte[8]
UNKNOWN	Unknown	

[1] BACnet OID (Object identifier) data type is included to allow interworking with BACnet (see [5] *ASHRAE 135-2004 Standard*). The format is described in the referenced standard.

149.20 Implementing a ZigBee Endpoint

OSGi services can also be exported as ZigBee endpoints to the local networks, in a way that is transparent to typical ZigBee devices endpoints. This allows developers to bridge legacy devices to ZigBee networks. A ZigBeeEndpoint MUST be registered with the following properties to export an OSGi service as a ZigBee endpoint:

- ZIGBEE_EXPORT – To indicate that the endpoint is an exportable endpoint.

An OSGi platform can be connected to multiple ZigBee networks. HOST_PID, PAN_ID and EXTENDED_PAN_ID are used to select the appropriate network. At least one of these properties MUST be specified. If provided, HOST_PID has priority over PAN_ID and EXTENDED_PAN_ID to identify the host that is targeted for export.

In addition, the ZigBeeEndpoint service MUST declare the same properties as an imported endpoint. The bundle registering endpoint services must make sure these properties are set accordingly or that none of these properties are set. In case a ZigBee host is not initialized yet or the base driver is not active on the OSGi framework, an endpoint implementation MAY not have any of the above identifiers.

If the Base Driver is active and at a ZigBee host is started, then the Base Driver makes an attempt to export the endpoint on the ZigBee network associated to the ZigBee HOST_PID, PAN_ID or EXTENDED_PAN_ID. The associated ZigBeeNode object MUST be one of the available ZigBeeHost objects. Every time an endpoint is registered or unregistered with both ZIGBEE_EXPORT and PAN_ID and/or EXTENDED_PAN_ID properties set, the associated ZigBeeHost service is modified accordingly (getEndpoints() returns a different array of ZigBeeEndpoint objects).

If - and only if - an error is detected on the properties of the ZigBee endpoint to be exported, then the Base Driver calls the notExported(ZigBeeException) method with a relevant ZigBeeException object as the input argument. The method SHOULD be called even if a ZigBee Host is not started.

The endpoint has to be registered with an ID that is unique. If the chosen ID already exists as a property of a local endpoint with the same host or if it already exists in an optional cache of the base driver, the base driver calls the notExported(ZigBeeException) method with the ZigBeeException object as the input argument with OSGI_EXISTING_ID error code. The base driver may keep IDs in a cache for endpoints that might come back in the registry. The range of potential IDs is 1-240 according to [1] *ZigBee Specification*.

The reader must note that a same ZigBeeEndpoint object cannot be registered several times with distinct PAN IDs since the getNodeAddress() method can only return one ZigBee node address.

If the PAN ID corresponds to more than one ZigBeeHost service, the ZigBeeEndpoint MUST define the Extended PAN ID property which uniquely identifies a ZigBee network. The base driver will call notExported(ZigBeeException) with the error code OSGI_MULTIPLE_HOSTS if the Extended PAN ID property is not properly defined in this specific situation.

Moreover, if the HOST PID corresponds to more than one ZigBeeHost, the base driver will also call notExported(ZigBeeException) with the error code OSGI_MULTIPLE_HOSTS.

149.21 Event API

Eventing is available in import and export situations:

- External events from the network must be dispatched to listeners inside the OSGi Service Platform. The ZigBee Base driver is responsible for mapping the network events to internal listener events.
- Implementations of ZigBee endpoints must send out events to local listeners. The ZigBee Base driver dispatches events to the network from its own listeners.

ZigBee events are sent using the whiteboard pattern, [6] *Listeners considered harmful: The whiteboard pattern*, in which a bundle interested in receiving the ZigBee events registers an object implementing the ZCLEventListener interface. The service MUST be registered with PAN_ID and/or EXTENDED_PAN_ID properties. These properties indicate the network targeted by the listener since an OSGi platform can host multiple ZigBee networks.

A filter can be set to limit the events for which a bundle is notified. The ZigBee Base driver must register a ZCLEventListener service for every attribute report configured in the configure reporting commands it receives from the network.

The filter refers to the combination of the properties registered with the ZCLEventListener service. Each ZCLEventListener MUST be registered with all the following mandatory properties:

- ID – (zigbee.cluster.id/Integer) Only events generated by endpoints matching a specific cluster are delivered.
- ID – (zigbee.attribute.id/Integer) Only events generated by endpoints matching a specific attribute are delivered.
- ATTRIBUTE_DATA_TYPE – (zigbee.attribute.datatype/Short) The Attribute data type field contains the data type of the attribute that is to be reported (see [2] *ZigBee Cluster Library Specification* 2.4.7.1.4 Attribute Data Type Field).

The optional properties are:

- IEEE_ADDRESS – (zigbee.node.ieee.address/BigInteger) Only events generated by endpoints matching the specific node are delivered.
- ENDPOINT_ID – (zigbee.endpoint.id/Short) Only events matching a specific endpoint are delivered.
- MIN_REPORT_INTERVAL – (zigbee.attribute.min.report.interval/Integer) The minimum interval, in seconds, between issuing reports of the specified attribute (see [2] *ZigBee Cluster Library Specification*. – 2.4.7.1.5).
- MAX_REPORT_INTERVAL – (zigbee.attribute.max.report.interval/Integer) The maximum interval, in seconds, between issuing reports of the specified attribute (see [2] *ZigBee Cluster Library Specification*. 2.4.7.1.6).

- REPORTABLE_CHANGE – (zigbee.attribute.reportable.change/Double) The minimum change to the attribute that will result in a report being issued. This property is mandatory if the data type is analog. If the data type is digital, the base driver will ignore it.

If the endpoint sets a timeout between two attribute reports, the notifyTimeOut(int) method is then called with the timeout argument. In the import situation, the base driver calls this method on the relevant listeners when it receives a configure reporting command with a set TIMEOUT_PERIOD field (see [2] *ZigBee Cluster Library Specification* 2.4.7 Configure Reporting Command). In the export situation, the local endpoint calls this method on relevant listeners and, in case the base driver is one of the notified listeners, it sends a configure reporting request with the appropriate TIMEOUT_PERIOD field to interested endpoints on the network.

A ZigBee event is represented by a ZigBeeEvent object.

If an event is generated by either the local endpoint or via the base driver for an external device, the notifyEvent(ZigBeeEvent) method is called on all registered ZCLEventListener services for which the source event matches the service properties. The way events must be delivered is the same as described in Delivering Events in the Life Cycle Layer chapter of the *OSGi Core Release 8* specification.

The ZigBee base driver SHOULD group subscriptions into one configure reporting request to the targeted ZigBee device. It SHOULD also notify every listener with respect to their specific expectations.

149.22 Monitoring Events and Sending Commands

In the example below, a button of the user interface monitors the state (on or off) of a smart plug and enables the user to switch the plug on and off. To monitor the plug state, a ZCLEventListener is registered with the properties related to the node, endpoint, cluster and attribute representing the plug and its state. When an appropriate event is sent on the network, the base driver (or a local endpoint implementer) notifies the listener. The listener then changes the state value shown by the button. When the user clicks on the button, a command is invoked on the plug.

```
public class UIOnOffButton implements ZCLEventListener {
    public UIOnOffButton(BigInteger ieeeAddress, Short endpointId, Integer
            clusterId, Integer attributeId, Short dataType,
            BundleContext bc) {
        Dictionary properties = new Hashtable();
        properties.put(ZigBeeNode.IEEE_ADDRESS, ieeeAddress);
        properties.put(ZigBeeEndpoint.ENDPOINT_ID, endpointId);
        properties.put(ZCLCluster.ID, clusterId);
        properties.put(ZCLAttribute.ID, attributeId);
        properties.put(ZCLEventListener.ATTRIBUTE_DATA_TYPE, dataType);
        // events will be filtered by the basedriver call notifyEvent() method
        // only when the event comes from a node, endpoint, cluster, attribute
        // matching these properties
        bc.registerService(ZCLEventListener.class.getName(), this, properties);
    }

    public void notifyEvent(ZigBeeEvent event) {
        // change the attribute value of the UICluster
        Object value = event.getValue();
        changeUIValue(value);
    }

    public void notifyTimeOut(int timeout) {
          log.info("Timeout notified");
    }

    public void onFailure(ZCLException e) {
          log.info("Failure registering the listener: " + e);
    }

    public void changeUIValue(Object value) {
    ....
    }

    public void onClick() {
        // the button has been clicked
        // get the ZCLCluster
        ServiceReference[] srs = bundleContext.getServiceReferences(
                          ZigBeeEndpoint.class.getName(),
                          "(&(" + ZigBeeNode.IEEE_ADDRESS + "=" + ieeeAddress
                          + ")(" + ZigBeeEndpoint.ENDPOINT_ID + "=" + endpointID
                          + "))");
        if (srs.length>0){
            ZCLCluster onOffCluster =
                ((ZigBeeEndpoint) bundleContext.getService(srs[0]))
                          .getServerCluster(ZCL_ONOFF_CLUSTER_ID);
            if (onOffCluster != null) {
                final Promise p = onOffCluster.invoke(new ToggleCommand());
                p.onResolve(new Runnable() {
                    public void run(){
                        try {
                            ZCLFrame frame = (ZCLFrame) p.getValue();
                            log.info("toggle command returned success.");
                        } catch (InvocationTargetException e) {
                            log.info("toggle command returned a failure: "
                                      + e + ".");
                        } catch (InterruptedException e) {
                             e.printStackTrace();
                        }
                    }
                });
            }
        }
    }

    class ToggleCommand implements ZCLFrame {
        ...
    }
...
}
```

149.23 ZCL Exception

The ZCLException extends the ZigBeeException. It holds information about the different ZigBee ZCL layers. Error codes specified by ZigBee Alliance are conveyed by the errorCode field of ZCLException objects.

149.24 ZDP Exception

The ZDPException extends the ZigBeeException. It holds information about the ZigBee ZDP layer. Error codes specified by ZigBee Alliance are conveyed by the errorCode field of ZDPException objects.

149.25 APS Exception

The APSException extends the ZigBeeException. It holds information about the ZigBee APS layer. Error codes specified by ZigBee Alliance are conveyed by the errorCode field of APSException objects.

149.26 ZigBee Exception

Some error codes are specified by the OSGi Working Group:

- OSGI_EXISTING_ID– another endpoint exists with the same ID.
- OSGI_MULTIPLE_HOSTS– several hosts exist for this PAN ID target or HOST_PID target.

149.27 ZCL Frame

The ZCLFrame contains a ZCLHeader, and a payload. It must used when invoking a command.

The ZCLHeader describes the header of a ZCLFrame.

The transaction id of each ZCLHeader must be managed by the base driver.

Only getters (not setters) are shared by client applications, the base driver and endpoint implementations. Therefore only getters are specified.

149.28 ZigBee Group

ZigBeeGroup enables group management (that is, it provides joinGroup(String) and leaveGroup(String) methods).

The creation of groups is made through the createGroupService(int) method.

A ZigBeeGroup service should be registered with the following property:

- ID – (zigbee.group.id/Integer) The 16-bit group address of the device.

And, the following ZigBeeEndpoint properties:

- DEVICE_CATEGORY

- INPUT_CLUSTERS
- HOST_PID

A ZigBeeGroup service enables the ZigBee groupcasting of command invocation thanks to the groupcast(int,ZCLFrame) and groupcast(int,ZCLFrame,String) methods. A groupcast message is received by the endpoints that are members of the targeted group.

149.29 ZigBee Networking

149.29.1 Logical node type

The ZigBee specification defines three types of ZigBee nodes on the network:

- ZigBee Coordinator (ZC) – The most capable device, the coordinator forms the root of the network. There is exactly one ZigBee coordinator in every network. It is able to store information about the network, to act as the Trust Center and repository for security keys. COORDINATOR represents the ZigBee coordinator.
- ZigBee Router (ZR) – A router is capable of extending a ZigBee network by routing data from other ZigBee devices. ROUTER represents a ZigBee router.
- ZigBee End Device (ZED) – An end device contains just enough functionality to talk to the parent node (either the coordinator or a router); it cannot relay data from other devices. ZED represents a ZigBee end device.

Every discovered ZigBeeNode on the network has a logical node type returned by calling the getLogicalType() method on the node's ZigBeeNodeDescriptor.

149.29.2 Network selection

The base driver provides a ZigBeeHost object for every available ZigBee local host. A ZigBee local host can represent a ZigBee chip on a USB dongle, a ZigBee built-in chip or a ZigBee Gateway Device (see [7] *ZigBee Gateway*). This object must be registered as a ZigBeeHost service. The ZigBeeHost interface has methods to start and stop the host, to store the networking configuration information (channel, channel mask, logical type, PAN ID, Extended PAN ID, security level, network key), and to open the network for devices to join it (permitJoin(short)).

ZigBeeHost also enables the broadcast of ZCL commands on a ZigBee network thanks to the broadcast(int,ZCLFrame) and broadcast(int,ZCLFrame,String) methods. Broadcasting enables the sending of a ZCL command to all clusters identified with an identifier of all endpoints available on the nodes of a ZigBee network within a number of hops defined by the broadcast radius of the coordinator (see the getBroadcastRadius() and setBroadcastRadius(short) methods).

In ZigBee networks, the coordinator must select a PAN identifier and a channel to start a network. After that, it behaves essentially like a router. The coordinator and routers can allow other devices to join the network and route data.

After an end device joins a router or coordinator, it is able to transmit or receive data through that router or coordinator. The router or coordinator that allowed an end device to join becomes the parent of the end device. Since the end device can sleep, the parent must be able to buffer or retain incoming data packets targeting the end device until the end device is able to wake up and receive the data.

149.29.3 Network coordination

When the ZigBeeHost is configured as the network coordinator, the getLogicalType() method on the node's ZigBeeNodeDescriptor MUST return COORDINATOR. The ZigBeeHost object will then be able to use the following operations for managing the network:

- updateNetworkChannel(byte) - Updates the network channel.
- setChannelMask(int) - Sets a new configured channel mask.
- refreshNetwork() – Requests the base driver to launch new discovery requests and refresh devices service registration according to current devices availability. This method is made mandatory since ZigBee specification allows devices not to notify the network when they leave it.

149.29.4 Networking considerations

The Network Address is a 16-bit address that is assigned by the coordinator when a node has joined a network and that must be unique for a given network in order for the node to be identified uniquely. ZigBeeNode provides getNetworkAddress() and getIEEEAddress() which returns device network address and device IEEE MAC address.

149.30 Security

149.30.1 Security management

ZigBee security is based on a 128-bit algorithm built on the security model provided by IEEE 802.15.4. ZigBee specification which defines the Trust Center.

The Trust Center is the device trusted by devices within a network to distribute keys for the purpose of network and end-to-end application configuration management. All members of the network shall recognize exactly one Trust Center, and there shall be exactly one Trust Center in each secure network.

The security of a network of ZigBee devices is based on link keys and a network key. Unicast communication between entities is secured by means of a 128-bit link key shared by two devices, one of those is normally the Trust Center. Broadcast communications are secured by means of a 128-bit network key shared among all devices in the network. The master key is only used as an initial shared secret between two devices when they perform the Key Establishment to generate Link Keys.

Security configuration is provided by the getSecurityLevel() method returning whether the security mode is activated or not on the ZigBee network.

A ZigBeeHost with a COORDINATOR logical node type will acts as a the Trust Center according to the ZigBee specification. It can also be any other device on the network. The Trust Center stores all the shared network keys. The getPreconfiguredLinkKey() method returns the network master key.

149.30.2 Conditional permission

When a bundle registers a ZigBeeEndpoint OSGi service, then the base driver exposes this endpoint on the outside ZigBee network and this endpoint has the ability to communicate with the other network devices. The base driver also provides an equivalent behavior when discovering a ZigBee endpoint from the outside network and exposing it as an OSGi Service in the OSGi Framework service registry. It is therefore recommended that ServicePermission[ZigBeeHost|ZigBeeEndpoint|ZCLEventListener, REGISTER|GET] be used sparingly and only for trusted bundles.

149.31 org.osgi.service.zigbee

Device Service Specification for ZigBee Technology.

This is the main package of this specification. It defines the interfaces that models the ZigBee concepts, like the ZigBee node and the ZigBee endpoint.

Each time a new ZigBee node is discovered, the driver will register a org.osgi.service.zigbee.ZigBeeNode service and then one org.osgi.service.zigbee.ZigBeeEndpoint service for each ZigBee endpoint discovered on the node.

org.osgi.service.zigbee.ZigBeeEndpoint interface provides the org.osgi.service.zigbee.ZigBeeEndpoint.getServerCluster(int) method to get an interface reference to a ZCLCluster object.

org.osgi.service.zigbee.ZCLCluster interface contains methods that directly maps to the ZCL profile-wide commands, like Read Attributes and Write Attributes, and allow the developer to forge its own commands and send them through the invoke() methods.

ZCL Attribute reportings are configured, registering a org.osgi.service.zigbee.ZCLEventListener, provided that this service is registered with the right service properties.

In addition to ZCL frames, the current specification allows also to send ZDP frames. Broadcasting and endpoint broadcasting is also supported for ZCL frames.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.zigbee; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.zigbee; version="[1.0,1.1)"
```

149.31.1 Summary

- `APSException` - This exception class is specialized for the APS errors.
- `ZCLAttribute` - This interface represents a ZCLAttribute.
- `ZCLAttributeInfo` - This interface provides information about the attribute, like its ZCL attribute ID, if it manufacturer specific and about its data type (see getDataType).
- `ZCLCluster` - This interface represents a ZCL Cluster.
- `ZCLCommandResponse` - A response event for a ZCLCommandResponseStream.
- `ZCLCommandResponseStream` - This type represents a stream of responses to a broadcast operation.
- `ZCLEventListener` - This interface represents a listener to events from ZigBee Device nodes.
- `ZCLException` - This class represents root exception for all the code related to ZigBee/ZCL.
- `ZCLFrame` - This interface models the ZigBee Cluster Library Frame.
- `ZCLHeader` - This interface represents the ZCL Frame Header.
- `ZCLReadStatusRecord` - This interface the reading result of ZCLCluster.readAttributes(ZCLAttributeInfo[]).
- `ZDPException` - This class represents root exception for all the code related to ZDP.
- `ZDPFrame` - This interface represents a ZDP frame.
- `ZDPResponse` - This type represents a successful ZDP invocation.
- `ZigBeeDataInput` - The ZigBeeDataInput interface is designed for converting a series of bytes in Java data types.
- `ZigBeeDataOutput` - The ZigBeeDataOutput interface is designed for converting Java data types into a series of bytes.
- `ZigBeeDataTypes` - This class contains the constants that are used internally by these API to represent the ZCL data types.
- `ZigBeeEndpoint` - This interface represents a ZigBee EndPoint.
- `ZigBeeEvent` - This interface represents events generated by a ZigBee Device node.

- `ZigBeeException` - This class represents root exception for all the code related to ZigBee.
- `ZigBeeGroup` - This interface represents a ZigBee Group.
- `ZigBeeHost` - This interface represents the machine that hosts the code to run a ZigBee device or client.
- `ZigBeeLinkQuality` - This interface represents an entry of the NeighborTableList.
- `ZigBeeNode` - This interface represents a ZigBee node, means a physical device that can communicate using the ZigBee protocol.
- `ZigBeeRoute` - This interface represents an entry of the RoutingTableList

149.31.2 public class APSException extends ZigBeeException

This exception class is specialized for the APS errors. See "Table 2.26 APS Sub-layer Status Values" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf.

149.31.2.1 public static final int ASDU_TOO_LONG = 65

A transmit request failed since the ASDU is too large and fragmentation is not supported.

149.31.2.2 public static final int DEFRAG_DEFERRED = 66

A received fragmented frame could not be defragmented at the current time.

149.31.2.3 public static final int DEFRAG_UNSUPPORTED = 67

A received fragmented frame could not be defragmented since the device does not support fragmentation.

149.31.2.4 public static final int ILLEGAL_REQUEST = 68

A parameter value was out of range.

149.31.2.5 public static final int INVALID_BINDING = 69

An APSME-UNBIND.request failed due to the requested binding link not existing in the binding table.

149.31.2.6 public static final int INVALID_GROUP = 70

An APSME-REMOVE-GROUP.request has been issued with a group identifier that does not appear in the group table.

149.31.2.7 public static final int INVALID_PARAMETER = 71

A parameter value was invalid or out of range.

149.31.2.8 public static final int NO_ACK = 72

An APSDE-DATA.request requesting acknowledged transmission failed due to no acknowledgment being received.

149.31.2.9 public static final int NO_BOUND_DEVICE = 73

An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to there being no devices bound to this device.

149.31.2.10 public static final int NO_SHORT_ADDRESS = 74

An APSDE-DATA.request with a destination addressing mode set to 0x03 failed due to no corresponding short address found in the address map table.

149.31.2.11 public static final int NOT_SUPPORTED = 75

An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to a binding table not being supported on the device.

149.31.2.12 public static final int SECURED_LINK_KEY = 76

An ASDU was received that was secured using a link key.

149.31.2.13 public static final int SECURED_NWK_KEY = 77

An ASDU was received that was secured using a network key.

149.31.2.14 public static final int SECURITY_FAIL = 78

An APSDE-DATA.request requesting security has resulted in an error during the corresponding security processing.

149.31.2.15 public static final int SUCCESS = 0

A request has been executed successfully.

149.31.2.16 public static final int TABLE_FULL = 79

An APSME-BIND.request or APSME.ADDGROUP. request issued when the binding or group tables, respectively, were full.

149.31.2.17 public static final int UNSECURED = 80

An ASDU was received without any security.

149.31.2.18 public static final int UNSUPPORTED_ATTRIBUTE = 81

An APSME-GET.request or APSMESET. request has been issued with an unknown attribute identifier.

149.31.2.19 public APSException(String errorDesc)

errorDesc exception an error description.

□ Creates a APSException containing only a description, but no error codes. If issued on this exception the getErrorCode() and getZigBeeErrorCode() methods return the UNKNOWN_ERROR constant.

149.31.2.20 public APSException(int errorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR if the actual error is not listed in this interface. In this case if the native ZigBee error code is known, it is preferred to use the APSException(int, int, String) constructor, passing UNKNOWN_ERROR as first parameter and the native ZigBee error as the second.

errorDesc An error description which explain the type of problem.

□ Creates a APSException containing a specific `errorCode`. Using this constructor with `errorCode` set to UNKNOWN_ERROR is equivalent to call APSException(String).

149.31.2.21 public APSException(int errorCode, int zigBeeErrorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR the actual error is not covered in this interface. In this case the `zigBeeErrorCode` parameter must be the actual status code returned by the ZigBee stack.

zigBeeErrorCode The actual APS status code or UNKNOWN_ERROR if this status is unknown.

errorDesc An error description which explain the type of problem.

□ Creates a APSException containing a specific `errorCode` or `zigBeeErrorCode`. Using this constructor with both the `errorCode` and `zigBeeErrorCode` set to UNKNOWN_ERROR is equivalent to call APSException(String).

149.31.3 public interface ZCLAttribute extends ZCLAttributeInfo

This interface represents a ZCLAttribute.

Its extends ZCLAttributeInfo to add methods to read and write the ZCL attribute from and to the ZigBee node with respectively the getValue() and setValue(Object) methods.

149.31.3.1 public static final String ID = "zigbee.attribute.id"

Property key for the optional attribute id of a ZigBee Event Listener.

149.31.3.2 public Promise<Object> getValue()

□ Gets the current value of the attribute.

As described in section *2.4.1.3 Effect on Receipt* of the ZCL specification, a *Read attributes* command can have the following status: ZCLException.SUCCESS, ZCLException.UNSUPPORTED_ATTRIBUTE, or ZCLException.INVALID_VALUE.

Returns A promise representing the completion of this asynchronous call. The response object returned by Promise.getValue() is the requested attribute value in the relevant Java data type (see getDataType() method and ZCLDataTypeDescription.getJavaDataType()) or in `byte[]` if getDataType() returns null. The response object is null if an ZCLException.UNSUPPORTED_ATTRIBUTE or ZCLException.INVALID_VALUE error occurs and the adequate ZCLException is returned by Promise.getFailure() .

149.31.3.3 public Promise<Void> setValue(Object value)

value the Java value to set.

□ Sets the current value of the attribute.

As described in section *2.4.3.3 Effect on Receipt* of the ZCL specification, a *Write attributes* command may return the following status: ZCLException.SUCCESS, ZCLException.UNSUPPORTED_ATTRIBUTE, ZCLException.INVALID_DATA_TYPE, ZCLException.READ_ONLY, ZCLException.INVALID_VALUE, or ZDPException.NOT_AUTHORIZED.

Returns A promise representing the completion of this asynchronous call. Promise.getFailure() returns null if the attribute value has been successfully written. The adequate ZigBeeException is returned otherwise.

149.31.4 public interface ZCLAttributeInfo

This interface provides information about the attribute, like its ZCL attribute ID, if it manufacturer specific and about its data type (see getDataType).

149.31.4.1 public static final String ID = "zigbee.attribute.id"

Property key for the optional attribute id of a ZigBee Event Listener.

149.31.4.2 public ZCLDataTypeDescription getDataType()

□ Returns the data type of this attribute.

Returns The attribute data type. It may be null if the data type is not retrievable (issue with read attribute and discover attributes commands).

149.31.4.3 public int getId()

□ Returns the ID of this attribute.

Returns the attribute identifier (that is, the attribute's ID).

149.31.4.4 public int getManufacturerCode()

□ Returns the manufacturer code of this attribute.

Returns The manufacturer code that defined this attribute, if the attribute does not belong to any manufacture extension then it returns -1.

149.31.4.5 public boolean isManufacturerSpecific()

□ Checks if the attribute is manufacturer specific.

Returns `true` if and only if this attribute is related to a manufacturer extension.

149.31.5 public interface ZCLCluster

This interface represents a ZCL Cluster. Along with methods to retrieve the cluster information, like its ID, it provides methods to asynchronously send commands to the cluster and other methods that wrap most of the ZCL general commands.

Every asynchronous method defined in this interface returns back its result through the use of a Promise.

149.31.5.1 public static final String DOMAIN = "zigbee.cluster.domain"

Property key for the optional cluster domain. A ZigBee Event Listener service can announce for what ZigBee clusters domains it wants notifications.

149.31.5.2 public static final String ID = "zigbee.cluster.id"

Property key for the optional cluster id. A ZigBee Event Listener service can announce for what ZigBee clusters it wants notifications.

149.31.5.3 public static final String NAME = "zigbee.cluster.name"

Property key for the optional cluster name. A ZigBee Event Listener service can announce for what ZigBee clusters it wants notifications.

149.31.5.4 public Promise<ZCLAttribute> getAttribute(int attributeId)

attributeId the ZCL attribute identifier.

□ Returns the cluster ZCLAttribute identifying that matches the given attributeId. ZCLCluster.getAttribute(int, int) method retrieves manufacturer-specific attributes.

Returns A promise representing the completion of this asynchronous call. In case of success in getting the attribute, the promise will be resolved with a ZCLAttribute instance. If attributeId do not exist in the cluster, then the promise fails with a ZCLException with status code ZCLException.UNSUPPORTED_ATTRIBUTE.

149.31.5.5 public Promise<ZCLAttribute> getAttribute(int attributeId, int code)

attributeId the ZCL attribute identifier

code the manufacturer code of the attribute to be retrieved. If -1 is used, the method behaves exactly like ZCLCluster.getAttribute(int)

□ Retrieves a ZCLAttribute object for a manufacturer specific attribute. If the `code` parameter is -1 it behaves like the ZCLCluster.getAttribute(int) and retrieves the non-manufacturer specific attribute `attributeId`.

Returns A Promise representing the completion of this asynchronous call. The promise will be resolved with the requested ZCLAttribute. If a command such as ZCL Read Attributes or Discover Attributes has already been called once by the ZigBee host, the Promise can be quickly resolved. The resolution may be longer the first time one of the ZCLCluster methods to get one or all attributes is successfully called. If attributeId do not exist in the cluster, then the promise fails with a ZCLException with status code ZCLException.UNSUPPORTED_ATTRIBUTE

149.31.5.6 public Promise<ZCLAttribute> getAttributes()

□ Returns an array of ZCLAttribute objects representing all this cluster's attributes.

This method returns only standard attributes. To retrieve manufacturer specific attributes use method ZCLCluster.getAttributes(int)

Returns A Promise representing the completion of this asynchronous call. The promise will be resolved with an array of ZCLAttribute objects.

149.31.5.7 public Promise<ZCLAttribute> getAttributes(int code)

code The the manufacturer code. Pass -1 to retrieve standard (that is, non-manufacturer specific) attributes.

□ Returns an array of ZCLAttribute objects representing all the specific manufacturer attributes available on the cluster.

This method behaves like the ZCLCluster.getAttributes() method if the passed value is -1.

Returns A Promise representing the completion of this asynchronous call. The promise will be resolved with an array of ZCLAttribute objects. If a command such as ZCL Read Attributes or Discover Attributes has already been called once by the ZigBee host, the Promise can be quickly resolved. The resolution may be longer the first time one of the ZCLCluster methods to get one or all attributes is successfully called.

149.31.5.8 public Promise<short> getCommandIds()

□ Returns an array of all the commandIds of the ZCLCluster.

This method is implemented for ZCL devices compliant version equal or later than 1.2 of the Home Automation Profile or other profiles that adds a general command that enables discovery of command identifiers. When the device implements a profile that does not support this feature, the promise fails with a ZCLException with code ZCLException.GENERAL_COMMAND_NOT_SUPPORTED.

Returns A Promise representing the completion of this asynchronous call. The promise will be resolved with `short[]` containing the command identifiers supported by the cluster.

149.31.5.9 public int getId()

□ Returns the identifier of this cluster.

Returns the cluster identifier.

149.31.5.10 public Promise<ZCLFrame> invoke(ZCLFrame frame)

frame The frame containing the command to issue.

□ Invokes a command on this cluster with a ZCLFrame. The returned promise provides the invocation response in an asynchronous way. The source endpoint is not specified in this method call. To send the appropriate message on the network, the base driver must generate a source endpoint. The latter must not correspond to any exported endpoint.

Returns A promise representing the completion of this asynchronous call. Promise.getValue() returns the response ZCLFrame.

149.31.5.11 public Promise<ZCLFrame> invoke(ZCLFrame frame, String exportedServicePID)

frame The frame containing the command to issue.

exportedServicePID : the source endpoint of the command request. In targeted situations, the source endpoint is the valid service PID of an exported endpoint.

□ Invokes a command on this cluster. This method is to be used by applications when the targeted device has to distinguish between source endpoints of the message. For instance, alarms cluster (see 3.11 Alarms Cluster in [ZCL]) generated events are differently interpreted if they come from the oven or from the intrusion alert system.

Returns A promise representing the completion of this asynchronous call. Promise.getValue() returns the response ZCLFrame.

149.31.5.12 public Promise<Map<Integer, ZCLReadStatusRecord>> readAttributes(ZCLAttributeInfo[] attributes)

attributes An array of ZCLAttributeInfo.

□ Reads a list of attributes by issuing a ZCL Read Attributes command. The attribute list is provided in terms of an array of ZCLAttributeInfo objects.

As described in section *2.4.1.3 Effect on Receipt* of the ZCL specification, a *Read Attributes* command results in a list of attribute status records comprising a mix of successful and unsuccessful attribute reads.

The method returns a promise. The object used to resolve the Promise is a `Map<Integer, ZCLReadStatusRecord>`. For each Map entry, the key contains the attribute identifier and the value, a ZigBee Read Attributes Status Record, which is made of the status of the read of this attribute, the ZigBee data type of the attribute and the attribute value in the corresponding Java wrapper type (or null in case of an unsupported attribute or in case of an invalid value). For attributes which data type serialization is not supported (that is, ZCLDataTypeDescription.getJavaDataType() returns null), the value is of type byte[].

When the list of attributes do not fit into a single ZCLFrame, ZigBee clusters truncate the list of attributes returned in the response. The client has to check the Map of results to send a new request for the attributes which values are missing. In export situations, the base driver may truncate the read attribute command response sent to networked devices in order to obey the rules.

NOTE: According to the ZigBee Specification all the attributes must be standard attributes or belong to the same manufacturer code, otherwise the promise must fail with a IllegalArgumentException exception.

Returns A promise representing the completion of this asynchronous call. The promise may fail with an IllegalArgumentException if the array size is 0 or if one of the array entries is `null` or not valid. An IllegalArgumentException is also thrown if some of ZCLAttributeInfo are manufacturer specific and other are standard, or even if there are mix of attributes with different manufacturer specific code. If the passed argument is `null` the promise must fail with a `NullPointerException`.

149.31.5.13 public Promise<Map<Integer, Integer>> writeAttributes(boolean undivided, Map<? extends ZCLAttributeInfo, ?> attributesAndValues)

undivided `true` if an undivided write attributes command is requested, `false` if not.

attributesAndValues A `Map<ZCLAttributeInfo, Object>` of attributes and values to be written. For ZCLAttributeInfo objects which serialization is not supported (that is, `getDataType().getJavaDataType()` returns null), the value must be of type byte[].

□ Writes a set of attributes on the cluster using the ZCL *Write Attributes* or the *Write Attributes Undivided* commands, according to the passed `undivided` parameter.

The promise resolves with a `Map<Integer, Integer>`. If all the attributes have been written successfully, the map is empty. In case of failure in writing specific attribute(s), the map is filled with entries related to those attributes. Every key is set with the id of an attribute that was not written successfully, every value with the status returned in the associated *write attribute response record* accordingly re-mapped to one of the constants defined in the ZCLException class.

According to the ZigBee Specification all the attributes must be standard attributes or, if manufacturer-specific they must have the same manufacturer code, otherwise an IllegalArgumentException occurs.

Returns A promise representing the completion of this asynchronous call. If resolved successfully the promise may return an empty `Map<Integer, Integer>`. Otherwise the map will be filled with the status information about the attributes that were not written. The key represents the attributeID and the value the status present in the corresponding attribute record returned by the ZCL Write Attributes response message. The original ZCL status values must be re-mapped to the list of status values listed in the ZCLException class. The promise may fail with an IllegalArgumentException if some of ZCLAttributeInfo are manufacturer specific and other are standard, or even if there are mix of attributes with different manufacturer specific code.

149.31.6 public interface ZCLCommandResponse

A response event for a ZCLCommandResponseStream.

149.31.6.1 public Promise<ZCLFrame> getResponse()

□ Returns a promise holding the response.

Returns A Promise holding the ZCLFrame response, or a failure exception if this is not a success response.

149.31.6.2 public boolean isEnd()

□ Checks if this is a terminal close event.

Returns `true` if this is a terminal close event.

149.31.7 public interface ZCLCommandResponseStream

This type represents a stream of responses to a broadcast operation. It can be closed by the client using the close method is called. The ZCLCommandResponseStream is used to process a stream of responses from a ZigBee network. Responses are consumed by registering a handler with forEach(Predicate). Responses received before a handler is registered are buffered until a handler is registered, or until the close method is called. A handler consumes events returning `true` to continue delivery. At some point the ZigBee service invocation will terminate event delivery by sending a close event (a ZCLCommandResponse which returns `true` from ZCLCommandResponse.isEnd(). After a close event the handler function will be dereferenced.

149.31.7.1 public void close()

□ Closes this response, indicating that no further responses are needed. Any buffered responses will be discarded, and a close event will be sent to a handler if it is registered.

149.31.7.2 public void forEach(Predicate<? super ZCLCommandResponse> handler)

handler A handler to process ZCLCommandResponse objects

□ Registers a handler that will be called for each of the received responses. Only one handler may be registered. Any responses that arrive before a handler is registered will be buffered and pushed into the handler when it is registered. If the handler returns `false` from its accept method then the handler will be released and no further events will be delivered. Any remaining buffered events will be discarded, and this object marked as closed. If the handler does not close the stream early then the ZigBee service implementation will eventually send a close event.

Throws `IllegalStateException`– if a handler has already been registered, or if this object has been closed (a ZCLCommandResponse which returns `true` from ZCLCommandResponse.isEnd(). After a close event the handler function will be dereferenced.

149.31.8 public interface ZCLEventListener

This interface represents a listener to events from ZigBee Device nodes.

149.31.8.1 public static final String ATTRIBUTE_DATA_TYPE = "zigbee.attribute.datatype"

Property key for the optional attribute data type of an attribute reporting configuration record, cf. ZCL Figure 2.16 Format of the Attribute Reporting Configuration Record.

149.31.8.2 public static final String MAX_REPORT_INTERVAL = "zigbee.attribute.max.report.interval"

Property key for the optional maximum interval, in seconds between issuing reports of the attribute. A ZigBee Event Listener service can declare the maximum frequency at which events it wants notifications.

149.31.8.3 public static final String MIN_REPORT_INTERVAL = "zigbee.attribute.min.report.interval"

Property key for the optional minimum interval, in seconds between issuing reports of the attribute. A ZigBee Event Listener service can declare the minimum frequency at which events it wants notifications.

149.31.8.4 public static final String REPORTABLE_CHANGE = "zigbee.attribute.reportable.change"

Property key for the optional maximum change to the attribute that will result in a report being issued. A ZigBee Event Listener service can declare the maximum frequency at which events it wants notifications.

149.31.8.5 public void notifyEvent(ZigBeeEvent event)

event a set of events.

□ Notifies the reception of an event. This method is called asynchronously.

149.31.8.6 public void notifyTimeOut(int timeout)

timeout the timeout in seconds.

□ Notifies that the timeout is elapsed. No event will be received in the interval.

149.31.8.7 public void onFailure(ZCLException e)

e the ZCLException.

□ Notifies that a failure has occurred.

That is, when either a ZCLException.UNSUPPORTED_ATTRIBUTE, ZCLException.UNREPORTABLE_TYPE, ZCLException.INVALID_VALUE, or ZCLException.INVALID_DATA_TYPE status occurs.

149.31.9 public class ZCLException extends ZigBeeException

This class represents root exception for all the code related to ZigBee/ZCL. The provided constants names, but not the values, maps to the ZCL error codes defined in the ZCL specification.

149.31.9.1 public static final int CALIBRATION_ERROR = 18

ZCL Calibration Error error code.

149.31.9.2 public static final int CLUSTER_COMMAND_NOT_SUPPORTED = 3

ZCL Cluster Command Not Supported error code.

149.31.9.3 public static final int DUPLICATE_EXISTS = 12

ZCL Duplicate Exists error code.

149.31.9.4 public static final int FAILURE = 1

ZCL Failure error code.

149.31.9.5 public static final int GENERAL_COMMAND_NOT_SUPPORTED = 4

ZCL General Command Not Supported error code.

149.31.9.6 public static final int HARDWARE_FAILURE = 16

HARDWARE_FAILURE - in this case, an additional exception describing the problem can be nested.

149.31.9.7 public static final int INSUFFICIENT_SPACE = 11

ZCL Insufficient Space error code.

149.31.9.8 public static final int INVALID_DATA_TYPE = 15

ZCL Invalid Data Type error code.

149.31.9.9 public static final int INVALID_FIELD = 7

ZCL Invalid Field error code.

149.31.9.10 public static final int INVALID_VALUE = 9

ZCL Invalid Value error code.

149.31.9.11 public static final int MALFORMED_COMMAND = 2

ZCL Malformed Command error code.

149.31.9.12 public static final int MANUF_CLUSTER_COMMAND_NOT_SUPPORTED = 5

ZCL Manuf Cluster Command Not Supported error code.

149.31.9.13 public static final int MANUF_GENERAL_COMMAND_NOT_SUPPORTED = 6

ZCL Manuf General Command Not Supported error code.

149.31.9.14 public static final int NOT_FOUND = 13

ZCL Not Found error code.

149.31.9.15 public static final int READ_ONLY = 10

ZCL Read Only error code.

149.31.9.16 public static final int SOFTWARE_FAILURE = 17

Software Failure error code - in this case, an additional exception describing the problem can be nested.

149.31.9.17 public static final int SUCCESS = 0

ZCL Success error code.

149.31.9.18 public static final int UNREPORTABLE_TYPE = 14

Unreportable Type error code.

149.31.9.19 public static final int UNSUPPORTED_ATTRIBUTE = 8

ZCL Unsupported Attribute error code.

149.31.9.20 public ZCLException(String errorDesc)

errorDesc exception error description.

□ Creates a ZCLException containing only a description, but no error codes. If issued on this exception the getErrorCode() and getZigBeeErrorCode() methods return the UNKNOWN_ERROR constant.

149.31.9.21 public ZCLException(int errorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR if the actual error is not listed in this interface. In this case if the native ZigBee error code is known, it is preferred to use the ZCLException(int, int, String) constructor, passing UNKNOWN_ERROR as first parameter and the native ZigBee error as the second.

errorDesc An error description which explain the type of problem.

□ Creates a ZCLException containing a specific errorCode. Using this constructor with errorCode set to UNKNOWN_ERROR is equivalent to call ZCLException(String).

149.31.9.22 public ZCLException(int errorCode, int zigBeeErrorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR the actual error is not covered in this interface. In this case the zigBeeErrorCode parameter must be the actual status code returned by the ZigBee stack.

zigBeeErrorCode The actual ZCL status code or UNKNOWN_ERROR if this status is unknown.

errorDesc An error description which explain the type of problem.

□ Creates a ZCLException containing a specific errorCode or zigBeeErrorCode. Using this constructor with both the errorCode and zigBeeErrorCode set to UNKNOWN_ERROR is equivalent to call ZCLException(String).

149.31.10 public interface ZCLFrame

This interface models the ZigBee Cluster Library Frame.

149.31.10.1 public byte[] getBytes()

□ Returns a byte array containing the raw ZCL frame, suitable to be sent on the wire. The returned byte array contains the whole ZCL Frame, including the ZCL Frame Header and the ZCL Frame payload.

Returns a byte array containing a raw ZCL frame, suitable to be sent on the wire. Any modifications issued on the returned array must not affect the internal representation of the ZCLFrame interface implementation.

149.31.10.2 public int getBytes(byte[] buffer)

buffer The buffer where to copy the raw ZCL frame.

□ Copy in the passed array the internal raw ZCLFrame.

Returns The actual number of bytes copied.

149.31.10.3 public ZigBeeDataInput getDataInput()

□ Returns ZigBeeDataInput for reading the ZCLFrame payload content. Every call to this method returns a different instance. The returned instances must not share the current position to the underlying ZCLFrame payload.

Returns a DataInput for the payload of the ZCLFrame. This method does not generate a copy of the payload.

Throws IllegalStateException– if the InputStream is not available.

149.31.10.4 public ZCLHeader getHeader()

□ Returns the header of this frame.

Returns the header of this frame.

149.31.10.5 public int getSize()

□ Retrieve the current size of the internal raw frame (that is the size of the byte[] that would be returned if calling the getBytes() method.

Returns The size of the raw ZCL frame.

149.31.11 public interface ZCLHeader

This interface represents the ZCL Frame Header.

149.31.11.1 public short getCommandId()

□ Returns the command identifier of this frame.

Returns the command identifier of this frame.

149.31.11.2 public short getFrameControlField()

□ Returns the Frame Control field of this frame.

Returns the frame control field of this frame.

149.31.11.3 public int getManufacturerCode()

□ Returns the manufacturer code of this frame.

Returns the manufacturer code if the ZCL Frame is manufacturer specific, otherwise returns -1.

149.31.11.4 public byte getSequenceNumber()

□ Returns the transaction Sequence Number of this frame.

Returns the transaction sequence number of this frame.

149.31.11.5 public boolean isClientServerDirection()

□ Checks the client server direction of the frame.

Returns the isClientServerDirection value.

149.31.11.6 public boolean isClusterSpecificCommand()

□ Checks the frame Type Sub-field of the frame control field.

Returns true if the frame control field states that the command is cluster specific. Returns false otherwise.

149.31.11.7 public boolean isDefaultResponseDisabled()

□ Checks if the default response is disabled.

Returns `true` if the ZCL Header Frame Control Field "Disable Default Response Sub-field" is 1. Returns `false` otherwise.

149.31.11.8 public boolean isManufacturerSpecific()

□ Checks if the frame is manufacturer specific.

Returns true if the ZCL frame is manufacturer specific (that is, the Manufacturer Specific Sub-field of the ZCL Frame Control Field is 1.

149.31.12 public interface ZCLReadStatusRecord

This interface the reading result of ZCLCluster.readAttributes(ZCLAttributeInfo[]).

149.31.12.1 public ZCLAttributeInfo getAttributeInfo()

□ Returns the ZCLAttributeInfo related to the reading operation.

Returns the ZCLAttributeInfo related to the reading operation.

149.31.12.2 **public ZigBeeException getFailure()**

□ Returns the potential failure of the reading operation.

Returns null in case of success, otherwise the ZigBeeException specifying the failing of the reading.

149.31.12.3 **public Object getValue()**

□ Returns the value of the related read attribute.

Returns null in case of failure or invalid data, otherwise the Java Object representing the ZigBee value.

149.31.13 public class ZDPException extends ZigBeeException

This class represents root exception for all the code related to ZDP.

See Table 2.137 ZDP Enumerations Description in ZIGBEE SPECIFICATION: 1_053474r17ZB_TSC-ZigBee-Specification.pdf.

149.31.13.1 **public static final int DEVICE_NOT_FOUND = 34**

The requested device did not exist on a device following a child descriptor request to a parent.

149.31.13.2 **public static final int INSUFFICIENT_SPACE = 42**

The device does not have storage space to support the requested operation.

149.31.13.3 **public static final int INV_REQUESTTYPE = 33**

The supplied request type was invalid.

149.31.13.4 **public static final int INVALID_EP = 35**

The supplied endpoint was equal to 0x00 or between 0xf1 and 0xff.

149.31.13.5 **public static final int NO_DESCRIPTOR = 41**

A child descriptor was not available following a discovery request to a parent.

149.31.13.6 **public static final int NO_ENTRY = 40**

The unbind request was unsuccessful due to the coordinator or source device not having an entry in its binding table to unbind.

149.31.13.7 **public static final int NO_MATCH = 39**

The end device bind request was unsuccessful due to a failure to match any suitable clusters.

149.31.13.8 **public static final int NOT_ACTIVE = 36**

The requested endpoint is not described by a simple descriptor.

149.31.13.9 **public static final int NOT_AUTHORIZED = 45**

The permissions configuration table on the target indicates that the request is not authorized from this device.

149.31.13.10 **public static final int NOT_PERMITTED = 43**

The device is not in the proper state to support the requested operation.

149.31.13.11 **public static final int NOT_SUPPORTED = 37**

The requested optional feature is not supported on the target device.

149.31.13.12 **public static final int SUCCESS = 0**

The requested operation or transmission was completed successfully.

149.31.13.13 public static final int TABLE_FULL = 44

The device does not have table space to support the operation.

149.31.13.14 public static final int TIMEOUT = 38

A timeout has occurred with the requested operation.

149.31.13.15 public ZDPException(String errorDesc)

errorDesc exception error description.

□ Creates a ZDPException containing only a description, but no error codes. If issued on this exception the getErrorCode() and getZigBeeErrorCode() methods return the UNKNOWN_ERROR constant.

149.31.13.16 public ZDPException(int errorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR if the actual error is not listed in this interface. In this case if the native ZigBee error code is known, it is preferred to use the ZDPException(int, int, String) constructor, passing UNKNOWN_ERROR as first parameter and the native ZigBee error as the second.

errorDesc An error description which explain the type of problem.

□ Creates a ZDPException containing a specific errorCode. Using this constructor with errorCode set to UNKNOWN_ERROR is equivalent to call ZDPException(String).

149.31.13.17 public ZDPException(int errorCode, int zigBeeErrorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR the actual error is not covered in this interface. In this case the zigBeeErrorCode parameter must be the actual status code returned by the ZigBee stack.

zigBeeErrorCode The actual ZDP status code or UNKNOWN_ERROR if this status is unknown.

errorDesc An error description which explain the type of problem.

□ Creates a ZDPException containing a specific errorCode or zigBeeErrorCode. Using this constructor with both the errorCode and zigBeeErrorCode set to UNKNOWN_ERROR is equivalent to call ZDPException(String).

149.31.14 public interface ZDPFrame

This interface represents a ZDP frame.

See Figure 2.19 Format of the ZDP Frame ZIGBEE SPECIFICATION: 1_053474r17ZB_TSC-ZigBee-Specification.pdf.

This interface MUST be implemented by the developer invoking the ZigBeeNode.invoke(int, int, ZDPFrame) method.

Notes:

- This interface hides on purpose the Transaction Sequence Number field because it MUST be handled internally by the ZigBee Base Driver
- The interface does not provide any method for writing the payload because the ZigBee Base Driver needs only to read the payload.

149.31.14.1 public ZigBeeDataInput getDataInput()

□ Returns the ZigBeeDataInput of the payload of this frame.

Returns the ZigBeeDataInput of the payload of the ZDPFrame. This method, in contrary to getPayload(), doesn't require to create a copy of the payload.

Throws IllegalStateException– if a ZigBeeDataInput stream cannot be returned because the underlying ZDPFrame implementation was not correctly initialized.

149.31.14.2 public byte[] getPayload()

□ Returns a copy of the payload of this frame.

Returns A copy of the payload of this frame.

149.31.15 public interface ZDPResponse

This type represents a successful ZDP invocation. Note that the underlying call may not have succeeded, The ZDPFrame frame must be introspected to identify the response from the ZigBeeNode.

149.31.15.1 public int getClusterId()

□ Returns the clusterId this response refers to.

Returns the clusterId this response refers to.

149.31.15.2 public ZDPFrame getFrame()

□ Returns the ZDPFrame containing the response.

Returns the ZDPFrame containing the response.

149.31.16 public interface ZigBeeDataInput

The ZigBeeDataInput interface is designed for converting a series of bytes in Java data types. The purpose of this interface is the same as the DataInput interface available in the standard Java library, with the difference that in this interface, byte ordering is little endian, whereas in the DataInput interface is big endian.

Each method provided by this interface read one or more bytes from the underlying stream, combine them, and return a Java data type. The pointer to the stream is then moved immediately after the last byte read. If this pointer past the available buffer bounds, a subsequent call to one of these methods will throw a EOFException.

149.31.16.1 public byte readByte() throws IOException

□ Reads a byte from the DataInput Stream.

Returns the byte read from the data input.

Throws EOFException– When the end of the input has been reached and there are no more data to read.

IOException– If an I/O error occurs.

149.31.16.2 public byte[] readBytes(int len) throws IOException

len the number of bytes to read.

□ Reads the specified amount of bytes from the underlying stream and return a copy of them. If the number of available bytes is less than the requested len, it throws an EOFException.

Returns return a copy of the bytes contained in the stream.

Throws EOFException– if there are not at least len bytes left on the data input.

IOException– If an I/O error occurs.

149.31.16.3 public double readDouble() throws IOException

□ Reads a number of type Double.

Returns a decoded double.

Throws EOFException– if there are not at least size 8 bytes left on the data input.

IOException– If an I/O error occurs.

149.31.16.4 public float readFloat(int size) throws IOException

size expected value for this parameter are 2 or 4 depending if reading ZigBeeDataTypes.FLOATING_SEMI or ZigBeeDataTypes.FLOATING_SINGLE.

□ Reads a number of type Float.

Returns The float number read from the data input.

Throws EOFException– if there are not at least size bytes left on the data input.

IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not in the allowed range.

149.31.16.5 public int readInt(int size) throws IOException

size the number of bytes that have to be read. Allowed values for this parameter are in the range [1, 4].

□ Reads an integer of the specified size. The sign bit of the size-bytes integer is left-extended. In other words if a readInt(2) is issued and the byte read are 0x01, 0x02 and 0xf0, the method returns 0xfff00201. For this reason if the 4 bytes read from the stream represent an unsigned value, to get the expected value the and bitwise operator must be used:

```
int u = readInt(3) & 0xffffff;
```

Returns the integer read from the data input.

Throws EOFException– When the end of the input has been reached and there are no more data to read.

IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not in the allowed range.

149.31.16.6 public long readLong(int size) throws IOException

size the number of bytes that have to be read. Allowed values for this parameter are in the range [1, 8].

□ Reads a certain amount of bytes and returns a long. The sign bit of the read size-bytes long is left-extended. In other words if a readLong(2) is issued and the byte read are 0x01 and 0xf0, the method returns 0xfffffffffffff001L. For this reason if the 2 bytes read from the stream represent an unsigned value, to get the expected value the and bitwise operator must be used:

```
long u = readLong(2) & 0xffff;
```

Returns The long value read from the data input.

Throws EOFException– if there are not at least size bytes left on the data input.

IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not in the allowed range.

149.31.17 public interface ZigBeeDataOutput

The ZigBeeDataOutput interface is designed for converting Java data types into a series of bytes. The purpose of this interface is the same as the DataOutput interface provided by Java, with the difference that in this interface, the generated bytes ordering is little endian, whereas in the DataOutput is big endian.

149.31.17.1 public void writeByte(byte value)

value The value to append.

□ Appends a byte to the data output.

To avoid losing information, the passed value must be in the range [-128, 127] for signed numbers and [0, 255] for unsigned numbers.

149.31.17.2 public void writeBytes(byte[] bytes, int length) throws IOException

bytes A buffer containing the bytes to append to the data output stream.

length The length in bytes that have to be actually appended.

□ Appends on the Data Output Stream a byte array. The byte array is written on the data output starting from the byte at index 0.

Throws IOException– If an I/O error occurs.

IllegalArgumentException– If the passed buffer is null or shorter than length bytes.

149.31.17.3 public void writeDouble(double value) throws IOException

value The double value to append.

□ Appends on the Data Output Stream a double value.

Throws IOException– If an I/O error occurs.

149.31.17.4 public void writeFloat(float value, int size) throws IOException

value The float value to append.

size The size in bytes that have to be actually appended. The size must be 2 for semi precision floats or 4 for standard precision floats (see the ZigBee Cluster Library specifications).

□ Appends on the Data Output Stream a float value.

Throws IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not within the allowed range.

149.31.17.5 public void writeInt(int value, int size) throws IOException

value The integer value to append

size The size in bytes that have to be actually appended. The size must be in the range [1,4].

□ Appends an int value to the data output.

To avoid losing information, according to the size argument, the passed long value if it represents a signed number must fit in the range [-2^(size * 8 - 1), -2^(size * 8 - 1) - 1].

For unsigned numbers it should fit in the range [0, -2^(size * 8) - 1].

For instance if size is 2 the correct range for signed numbers is [0xffff8000, 0x7fff] (that is, [-32768, +32767]), whereas for unsigned numbers is [0L, 0xffff].

Although this method allows write even 1 byte of the passed int value, it is suggested to use the writeByte(byte) because this latter could be implemented in a more efficient way.

Throws IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not within the allowed range.

149.31.17.6 public void writeLong(long value, int size) throws IOException

value The long value to append

size The size in bytes that have to be actually appended. The size must be in the range [1,8].

□ Appends a long to the data output.

To avoid losing information, according to the size argument, the passed long value if it represents a signed number must fit in the range [-2^(size * 8 - 1), -2^(size * 8 - 1) - 1].

For unsigned numbers it should fit in the range [0, -2^(size * 8) - 1].

For instance if size is 3 the correct range for signed numbers is [0xffffffffff800000L, 0x7fffffL] (that is, [-21474836448, +2147483647]), whereas for unsigned numbers is [0L, 0xffffffL].

Although this method allows write even 1 byte of the passed long value, it is suggested to use the writeByte(byte) because this latter could be implemented in a more efficient way.

Throws IOException– If an I/O error occurs.

IllegalArgumentException– If the passed size is not within the allowed range.

149.31.18 public class ZigBeeDataTypes

This class contains the constants that are used internally by these API to represent the ZCL data types.

These constants do not match the values used in the ZigBee specification, but follow the rules below:

- *bit 0-3*: if bit 6 is one, these bits represents the size of the data type in bytes.
- *bit 6*: if set to 1 bits 0-3 represents the size of the data type in bytes.

Related documentation: [1] ZigBee Cluster Library specification, Document 075123r04ZB, May 29, 2012.

149.31.18.1 public static final short ARRAY = 16

According to ZigBee Cluster Library [1], an Array is an ordered sequence of zero or more elements, all of the same data type. This data type may be any ZCL defined data type, including Array, Structure, Bag or Set. The total nesting depth is limited to 15.

149.31.18.2 public static final short ATTRIBUTE_ID = 6

The type of an attribute identifier.

149.31.18.3 public static final short BACNET_OID = 7

According to ZigBee Cluster Library [1], the BACnet OID data type is included to allow interworking with BACnet. The format is described in the referenced standard.

149.31.18.4 public static final short BAG = 19

According to ZigBee Cluster Library [1], a Bag behaves exactly the same as a Set, except that two elements may have the same value.

149.31.18.5 public static final short BITMAP_16 = 89

Bitmap16-bit

149.31.18.6 public static final short BITMAP_24 = 90

Bitmap 24-bit

149.31.18.7 public static final short BITMAP_32 = 91

Bitmap 32-bit

149.31.18.8 public static final short BITMAP_40 = 92

Bitmap 40-bit

149.31.18.9 public static final short BITMAP_48 = 93

Bitmap 48-bit

149.31.18.10 **public static final short BITMAP_56 = 94**

Bitmap 56-bit

149.31.18.11 **public static final short BITMAP_64 = 95**

Bitmap 64-bit

149.31.18.12 **public static final short BITMAP_8 = 88**

According to ZigBee Cluster Library [1], the Bitmap type holds logical values, one per bit, depending on its length. There is no value that represents an invalid value of this type. The Bitmap type is defined with several sizes: 8, 16, 24, 32, 40, 48, 56 and 64 bits.

149.31.18.13 **public static final short BOOLEAN = 1**

According to ZigBee Cluster Library [1], the Boolean type represents a logical value, either FALSE (0x00) or TRUE (0x01). The value 0xff represents an invalid value of this type. All other values of this type are forbidden.

149.31.18.14 **public static final short CHARACTER_STRING = 121**

According to ZigBee Cluster Library [1], the Character String data type contains data octets encoding characters according to the language and character set field of the complex descriptor.

149.31.18.15 **public static final short CLUSTER_ID = 5**

The type of a cluster identifier.

149.31.18.16 **public static final short DATE = 3**

The Date data type format is specified in section 2.5.2.20 of ZigBee Cluster Specification [1].

149.31.18.17 **public static final short ENUMERATION_16 = 113**

Enumeration 16-bit

149.31.18.18 **public static final short ENUMERATION_8 = 112**

According to ZigBee Cluster Library [1], the Enumeration type represents an index into a lookup table to determine the final value. The values 0xff and 0xffff represent invalid values of the 8-bit and 16-bit types respectively.

149.31.18.19 **public static final short FLOATING_DOUBLE = 250**

According to ZigBee Cluster Library [1], the format of the double precision data type is based on the IEEE 754 standard for binary floating-point arithmetic.

149.31.18.20 **public static final short FLOATING_SEMI = 248**

According to ZigBee Cluster Library [1], the ZigBee semi-precision number format is based on the IEEE 754 standard for binary floating-point arithmetic.

149.31.18.21 **public static final short FLOATING_SINGLE = 249**

According to ZigBee Cluster Library [1], the format of the single precision data type is based on the IEEE 754 standard for binary floating-point arithmetic.

149.31.18.22 **public static final short GENERAL_DATA_16 = 81**

General Data 16-bit

149.31.18.23 **public static final short GENERAL_DATA_24 = 82**

General Data 24-bit

149.31.18.24 public static final short GENERAL_DATA_32 = 83

General Data 32-bit

149.31.18.25 public static final short GENERAL_DATA_40 = 84

General Data 40-bit

149.31.18.26 public static final short GENERAL_DATA_48 = 85

General Data 48-bit

149.31.18.27 public static final short GENERAL_DATA_56 = 86

General Data 56-bit

149.31.18.28 public static final short GENERAL_DATA_64 = 87

General Data 64-bit

149.31.18.29 public static final short GENERAL_DATA_8 = 80

According to ZigBee Cluster Library [1], the General Data type may be used when a data element is needed but its use does not conform to any of other types. The General Data type is defined with several sizes: 8, 16, 24, 32, 40, 48, 56 and 64 bits.

149.31.18.30 public static final short IEEE_ADDRESS = 8

According to ZigBee Cluster Library [1], the IEEE Address data type is a 64-bit IEEE address that is unique to every ZigBee device. A value of 0xffffffffffffffff indicates that the address is unknown.

149.31.18.31 public static final short LONG_CHARACTER_STRING = 123

According to ZigBee Cluster Library [1], the Long Character String data type contains data octets encoding characters according to the language and character set field of the complex descriptor.

149.31.18.32 public static final short LONG_OCTET_STRING = 122

According to ZigBee Cluster Library [1], the Long Octet String data type contains data in application-defined formats.

149.31.18.33 public static final short NO_DATA = 0

According to ZigBee Cluster Library [1], the no data type represents an attribute with no associated data.

149.31.18.34 public static final short OCTET_STRING = 120

According to ZigBee Cluster Library [1], the Octet String data type contains data in application-defined formats.

149.31.18.35 public static final short SECURITY_KEY_128 = 9

According to ZigBee Cluster Library [1], the 128-bit Security Key data type is for use in ZigBee security, and may take any 128-bit value.

149.31.18.36 public static final short SET = 18

According to ZigBee Cluster Library [1], a Set is a collection of elements with no associated order. Each element has the same data type, which may be any ZCL defined data type, including Array, Structure, Bag or Set. The nesting depth is limited to 15.

149.31.18.37 public static final short SIGNED_INTEGER_16 = 225

Signed Integer 16-bit

149.31.18.38 public static final short SIGNED_INTEGER_24 = 226

Signed Integer 24-bit

149.31.18.39 public static final short SIGNED_INTEGER_32 = 227

Signed Integer 32-bit

149.31.18.40 public static final short SIGNED_INTEGER_40 = 228

Signed Integer 40-bit

149.31.18.41 public static final short SIGNED_INTEGER_48 = 229

Signed Integer 48-bit

149.31.18.42 public static final short SIGNED_INTEGER_56 = 230

Signed Integer 56-bit

149.31.18.43 public static final short SIGNED_INTEGER_64 = 231

Signed Integer 64-bit

149.31.18.44 public static final short SIGNED_INTEGER_8 = 224

According to ZigBee Cluster Library [1], the Signed Integer type represents a signed integer with a decimal range of -(2^7-1) to 2^7-1, - (2^15-1) to 2^15-1, -(2^23-1) to 2^23-1, -(2^31-1) to 2^31-1, -(2^39-1) to 2^39-1, -(2^47-1) to 2^47-1, -(2^55-1) to 2^55-1, or -(2^63-1) to 2^63-1, depending on its length. The values that represents an invalid value of this type are 0x80, 0x8000, 0x800000, 0x80000000, 0x8000000000, 0x800000000000, 0x80000000000000 and 0x8000000000000000 respectively. This type is defined with several sizes: 8, 16, 24, 32, 40, 48, 56 and 64 bits.

149.31.18.45 public static final short STRUCTURE = 17

According to ZigBee Cluster Library [1], a Structure is an ordered sequence of elements, which may be of different data types. Each data type may be any ZCL defined data type, including Array, Structure, Bag or Set. The total nesting depth is limited to 15.

149.31.18.46 public static final short TIME_OF_DAY = 2

The Time of Day data type format is specified in section 2.5.2.19 of ZCL specification [1].

149.31.18.47 public static final short UNKNOWN = 255

The UNKNOWN type is used when the data type is unknown.

149.31.18.48 public static final short UNSIGNED_INTEGER_16 = 97

Unsigned Integer 16-bit

149.31.18.49 public static final short UNSIGNED_INTEGER_24 = 98

Unsigned Integer 24-bit

149.31.18.50 public static final short UNSIGNED_INTEGER_32 = 99

Unsigned Integer 32-bit

149.31.18.51 public static final short UNSIGNED_INTEGER_40 = 100

Unsigned Integer 40-bit

149.31.18.52 public static final short UNSIGNED_INTEGER_48 = 101

Unsigned Integer 48-bit

149.31.18.53 public static final short UNSIGNED_INTEGER_56 = 102

Unsigned Integer 56-bit

149.31.18.54 public static final short UNSIGNED_INTEGER_64 = 103

Unsigned Integer 64-bit

149.31.18.55 public static final short UNSIGNED_INTEGER_8 = 96

According to ZigBee Cluster Library [1], the Unsigned Integer type represents an unsigned integer with a decimal range of 0 to 2^8-1, 0 to 2^16-1, 0 to 2^24-1, 0 to 2^32-1, 0 to 2^40-1, 0 to 2^48-1, 0 to 2^56-1, or 0 to 2^64-1, depending on its length. The values that represents an invalid value of this type are 0xff, 0xffff, 0xffffff, 0xffffffff, 0xffffffffff, 0xffffffffffff, 0xffffffffffffff and 0xffffffffffffffff respectively. This type is defined with several sizes: 8, 16, 24, 32, 40, 48, 56 and 64 bits.

149.31.18.56 public static final short UTC_TIME = 4

According to ZigBee Cluster Library [1], UTCTime is an unsigned 32-bit value representing the number of seconds since 0 hours, 0 minutes, 0 seconds, on the 1st of January, 2000 UTC (Universal Coordinated Time). The value that represents an invalid value of this type is 0xffffffff.

149.31.19 public interface ZigBeeEndpoint

This interface represents a ZigBee EndPoint. A ZigBeeEndpoint must be registered as a OSGi service with ZigBeeNode.IEEE_ADDRESS, and ZigBeeEndpoint.ENDPOINT_ID properties.

149.31.19.1 public static final String DEVICE_CATEGORY = "ZigBee"

Constant used by all ZigBee devices indicating the device category. It is a **mandatory** service property for this service.

149.31.19.2 public static final String DEVICE_ID = "zigbee.device.id"

Property containing the application device identifier. This identifier is also contained in the ZigBee Simple Descriptor. This property is of type Integer.

It is **mandatory** property for this service.

149.31.19.3 public static final String DEVICE_VERSION = "zigbee.device.version"

Property containing the application device version. The application device version is also contained in the ZigBee endpoint Simple Descriptor. This property is of type Byte.

It is **mandatory** property for this service.

149.31.19.4 public static final String ENDPOINT_ID = "zigbee.endpoint.id"

Property containing the EndPoint ID of the device. This property is of type Short and its value must be in the range allowed by the ZigBee specifications for Zigbee endpoints identifiers.

It is **mandatory** service property for ZigBeeEndpoint services.

149.31.19.5 public static final String HOST_PID = "zigbee.endpoint.host.pid"

Property containing the ZigBeeHost's pid. This property is of type String.

The ZigBee local host identifier is intended to uniquely identify the ZigBee local host, since there could be many hosts on the same platform.

All the endpoints that belong to a specific network MUST specify the value of the associated host pid. It is mandatory for imported endpoints, optional for exported endpoints.

149.31.19.6 public static final String INPUT_CLUSTERS = "zigbee.endpoint.clusters.input"

Property containing a list of input clusters. This list is contained also in the ZigBee Simple Descriptor returned by the ZigBeeEndpoint service. This property is of type int[].

It is **mandatory** service property for this service.

149.31.19.7 public static final String OUTPUT_CLUSTERS = "zigbee.endpoint.clusters.output"

Property containing a list of output clusters. This list is contained also in the ZigBee Simple Descriptor of the ZigBeeEndpoint service. This property is of type int[].

It is a **mandatory** service property for this service.

149.31.19.8 public static final String PROFILE_ID = "zigbee.device.profile.id"

Property containing the application profile identifier also contained in the ZigBee Simple Descriptor. This property is of type Integer.

It is **mandatory** service property for this service.

149.31.19.9 public static final String ZIGBEE_EXPORT = "zigbee.export"

Property used to mark if a ZigBeeEndPoint service is an exported one or not. Imported endpoints do not have this property set. This service property requires no specific values.

149.31.19.10 public Promise<Void> bind(String servicePid, int clusterId)

servicePid the PID of the endpoint to bind to

clusterId the cluster identifier to bind to

□ Adds the following entry in the *Binding Table* of the device:

`this.getNodeAddress(), this.getId(), clusterId, device.getNodeAddress(), device.getId()`

As described in "Table 2.7 APSME-BIND.confirm Parameters" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a binding request can have the following results: APSException.SUCCESS, APSException.ILLEGAL_REQUEST, APSException.TABLE_FULL, APSException.NOT_SUPPORTED.

Returns A promise representing the completion of this asynchronous call. Promise.getFailure() returns null if the cluster has been successfully bound. The adequate ZigBeeEndpoint is returned otherwise.

149.31.19.11 public Promise<List<String>> getBoundEndPoints(int clusterId)

clusterId the cluster identifier of the targeted bindings.

□ Returns bound endpoints (identified by their service PIDs) on a specific cluster ID. It is implemented on the base driver with Mgmt_Bind_req command. It is implemented without a command request in local endpoints.

As described in "Table 2.129 Fields of the Mgmt_Bind_rsp Command" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a Mgmt_Bind_rsp command can have the following status: APSException.NOT_SUPPORTED or any status code returned from the APSME-GET.confirm primitive (see APSException).

Returns A promise representing the completion of this asynchronous call. Promise.getValue() returns a List of the bound endpoint service PIDs if the command is successful. The response object is null and the adequate APSException is returned by Promise.getFailure() otherwise.

149.31.19.12 public ZCLCluster getClientCluster(int clientClusterId)

clientClusterId The client(output) cluster identifier.

□ Returns the client cluster identified by the cluster identifier.

Returns the client(output) cluster identified by the cluster identifier, or null if the given id is not listed in the simple descriptor.

Throws `IllegalArgumentException` – If the passed argument is outside the range [0, 0xffff].

149.31.19.13 public ZCLCluster[] getClientClusters()

□ Returns an array of client (output) clusters.

Returns an array of client (output) clusters, returns an empty array if does not provides any clients clusters.

149.31.19.14 public short getId()

□ Returns the identifier of this endpoint, that is the Endpoint ID.

Returns the identifier of this endpoint, value ranges from 1 to 240.

149.31.19.15 public BigInteger getNodeAddress()

□ Returns the IEEE Address of the node containing this endpoint.

Returns the IEEE Address of the node containing this endpoint.

149.31.19.16 public ZCLCluster getServerCluster(int serverClusterId)

serverClusterId The server(input) cluster identifier.

□ Returns the server (input) cluster identified by the given identifier.

Returns the server (input) cluster identified by the given identifier, or null if the given id is not listed in the simple descriptor.

Throws `IllegalArgumentException`– If the passed argument is outside the range [0, 0xffff].

149.31.19.17 public ZCLCluster[] getServerClusters()

□ Returns an array of server (input) clusters.

Returns an array of server (input) clusters, returns an empty array if it does not provide any server cluster.

149.31.19.18 public Promise<ZigBeeSimpleDescriptor> getSimpleDescriptor()

□ Returns the simple descriptor of this endpoint. As described in "Table 2.93 Fields of the Simple_Desc_rsp Command" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a simple_decr request can have the following status: ZDPException.SUCCESS, ZDPException.INVALID_EP, ZDPException.NOT_ACTIVE, ZDPException.DEVICE_NOT_FOUND, ZDPException.INV_REQUESTTYPE or ZDPException.NO_DESCRIPTOR.

Returns A promise representing the completion of this asynchronous call. Promise.getValue() returns the node simple descriptor ZigBeeSimpleDescriptor in case of success and Promise.getFailure() returns the adequate ZDPException otherwise.

149.31.19.19 public void notExported(ZigBeeException e)

e A device ZigBeeException the occurred exception.

□ Notifies that the base driver is unable to export this endpoint. This method is called by the base driver and used to give details about issues preventing the export of an endpoint.

149.31.19.20 public Promise<Void> unbind(String servicePid, int clusterId)

servicePid The pid of the service to unbind.

clusterId The cluster identifier to unbind.

□ Removes the following entry in the *Binding Table* of the device if it exists:

`this.getNodeAddress(), this.getId(), clusterId, device.getNodeAddress(), device.getId()`

As described in "Table 2.9 APSME-UNBIND.confirm Parameters" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, an unbind request can have the following results: APSException.SUCCESS, APSException.ILLEGAL_REQUEST, APSException.INVALID_BINDING.

Returns A promise representing the completion of this asynchronous call. Promise.getFailure() returns null if the cluster has been successfully bound. The adequate APSException is returned otherwise.

149.31.20 public interface ZigBeeEvent

This interface represents events generated by a ZigBee Device node.

149.31.20.1 public int getAttributeId()

□ Returns the attribute identifier (that is, the attribute's ID).

Returns the attribute identifier (that is, the attribute's ID).

149.31.20.2 public int getClusterId()

□ Returns the cluster id associated to this ZigBeeEvent.

Returns the cluster id.

149.31.20.3 public short getEndpointId()

□ Returns the endpoint identifier.

Returns the endpoint identifier.

149.31.20.4 public BigInteger getIEEEAddress()

□ Returns the ZigBee device node IEEE Address.

Returns the ZigBee device node IEEE Address.

149.31.20.5 public Object getValue()

□ Returns an object containing the new value of the related ZigBee attribute.

Returns an object containing the new value for the ZigBee attribute that has changed.

149.31.21 public class ZigBeeException extends RuntimeException

This class represents root exception for all the code related to ZigBee. The provided constants names, but not the values.

149.31.21.1 protected final int errorCode

The error code associated to this exception.

See Also getErrorCode()

149.31.21.2 public static final int OSGI_EXISTING_ID = 48

The error code used when another endpoint exists with the same ID.

149.31.21.3 public static final int OSGI_MULTIPLE_HOSTS = 49

The error code used when several hosts exist for this PAN ID target or HOST_PID target.

149.31.21.4 public static final int TIMEOUT = 50

The error code used when the timeout of ZigBee asynchronous exchange is reached.

149.31.21.5 public static final int UNKNOWN_ERROR = -1

This error code is used if the ZigBee error returned is not covered by this API specification.

149.31.21.6 protected final int zigBeeErrorCode

The actual error code returned by the ZigBee node.

See Also ZigBeeException.getZigBeeErrorCode()

149.31.21.7 public ZigBeeException(String errorDesc)

errorDesc exception error description.

□ Creates a ZigBeeException containing only a description, but no error codes. If issued on this exception the getErrorCode() and getZigBeeErrorCode() methods return the UNKNOWN_ERROR constant.

149.31.21.8 public ZigBeeException(int errorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR if the actual error is not listed in this interface.

errorDesc An error description which explain the type of problem.

□ Creates a ZigBeeException containing a specific errorCode. Using this constructor with errorCode set to UNKNOWN_ERROR is equivalent to call ZigBeeException(String).

149.31.21.9 public ZigBeeException(int errorCode, int zigBeeErrorCode, String errorDesc)

errorCode One of the error codes defined in this interface or UNKNOWN_ERROR the actual error is not covered in this interface.

zigBeeErrorCode The actual status code or UNKNOWN_ERROR if this status is unknown.

errorDesc An error description which explain the type of problem.

□ Creates a ZigBeeException containing a specific errorCode or zigBeeErrorCode. Using this constructor with both the errorCode and zigBeeErrorCode set to UNKNOWN_ERROR is equivalent to call ZigBeeException(String).

149.31.21.10 public int getErrorCode()

□ Returns the error code.

Returns the error code.

149.31.21.11 public int getZigBeeErrorCode()

□ Returns the potential ZigBee error code.

Returns One of the error codes defined above. If the returned error code is UNKNOWN_ERROR and the hasZigBeeErrorCode() returns true then the getZigBeeErrorCode() provides the actual ZigBee error code returned by the device.

149.31.21.12 public boolean hasZigBeeErrorCode()

□ Checks if this exception has a ZigBee error code.

Returns true if the ZigBeeException convey also the actual error code returned by the ZigBee stack.

149.31.22 public interface ZigBeeGroup

This interface represents a ZigBee Group.

No Implement Consumers of this API must not implement this interface

149.31.22.1 public static final String ID = "zigbee.group.id"

Key of the String containing the Group Address of the device.

It is a **mandatory** property for this service.

149.31.22.2 public int getGroupAddress()

□ Returns the 16-bit group address.

Returns the 16-bit group address.

149.31.22.3 public ZCLCommandResponseStream groupcast(int clusterId, ZCLFrame frame)

clusterId a cluster identifier.

frame a command frame sequence.

□ Sends a ZCL frame to the group represented by this service. The returned stream will provide the invocation response(s) in an asynchronous way.

The source endpoint is not specified in this method call. To send the appropriate message on the network, the base driver must generate a source endpoint. The latter must not correspond to any exported endpoint.

Returns a ZCLCommandResponseStream to collect every ZCL frame one after the other in case of multiple responses.

149.31.22.4 public ZCLCommandResponseStream groupcast(int clusterId, ZCLFrame frame, String exportedServicePID)

clusterId a cluster identifier.

frame a command frame sequence.

exportedServicePID : the source endpoint of the command request. In targeted situations, the source endpoint is the valid service PID of an exported endpoint.

□ Sends a ZCL frame to the ZigBee group represented by this service. The returned stream will provide the invocation response(s) in an asynchronous way.

This method is to be used by applications when the targeted device has to distinguish between source endpoints of the message. For instance, alarms cluster (see 3.11 Alarms Cluster in [ZCL]) generated events are differently interpreted if they come from the oven or from the intrusion alert system.

Returns a ZCLCommandResponseStream to collect every ZCL frame one after the other in case of multiple responses.

149.31.22.5 public Promise<Void> joinGroup(String pid)

pid String representing the service PID of the ZigBeeEndpoint to add to this Group.

□ Requests an endpoint to join this group. This method may be invoked on exported and imported endpoints. In the former case, the ZigBee Base Driver should rely on the *APSME-ADD-GROUP* API defined by the ZigBee Specification, or it will use the proper commands of the *Groups* cluster of the ZigBee Specification Library. As described in "Table 2.15 APSME-ADD-GROUP.confirm Parameters" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, an add_group request can have the following status: APSException.SUCCESS, APSException.INVALID_PARAMETER or APSException.TABLE_FULL. When the joining is performed remotely on an imported ZigBeeEndpoint, it may also fail because the command is not supported by the remote endpoint, or because the remote device cannot perform the operation at the moment (see ZCLException).

Returns A promise representing the completion of this asynchronous call. Promise.getFailure() returns null if the cluster has been successfully bound. The adequate ZigBeeException is returned otherwise.

149.31.22.6 public Promise<Void> leaveGroup(String pid)

pid String representing the service PID of the ZigBeeEndpoint to remove from this Group.

□ Requests an endpoint to leave this group. This method may be invoked on exported and imported endpoints. In the former case, the ZigBee Base Driver should rely on the *APSME-REMOVE-GROUP* API defined by the ZigBee Specification, or it will use the proper commands of the *Groups* cluster of the ZigBee Specification Library. As described in "Table 2.17 APSME-REMOVE-GROUP.confirm Parameters" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a remove_group request can have the following status: APSException.SUCCESS, APSException.INVALID_GROUP or

APSException.INVALID_PARAMETER. When the command is invoked remotely on an imported ZigBeeEndpoint, it may also fail because the command is not supported by the remote endpoint, or because the remote device cannot perform the operation at the moment (see ZCLException).

Returns A promise representing the completion of this asynchronous call. Promise.getFailure() returns null if the cluster has been successfully bound. The adequate ZigBeeException is returned otherwise.

149.31.23 public interface ZigBeeHost extends ZigBeeNode

This interface represents the machine that hosts the code to run a ZigBee device or client. This machine is, for example, the ZigBee chip/dongle that is controlled by the base driver (below/under the OSGi execution environment).

ZigBeeHost is more than a ZigBeeNode.

It must be registered as a OSGi service.

Even if not specified explicitly in the javadoc, any method of this interface must throw an IllegalArgumentException exception if a or one of the passed arguments has a value not admitted by the method.

No Implement Consumers of this API must not implement this interface

149.31.23.1 public static final short UNLIMITED_BROADCAST_RADIUS = 255

Value constant to set an unlimited broadcast radius.

149.31.23.2 public ZCLCommandResponseStream broadcast(int clusterID, ZCLFrame frame)

clusterID The cluster ID this ZCL frame must be sent to.

frame A ZCL Frame.

□ Broadcasts a ZCL frame to the cluster ID of all the nodes of the ZigBee network. The setBroadcastRadius(short) method, may be used to limit the broadcast radius used in the subsequent broadcast calls.

Returns a response stream instance that collects and allows the caller to be asynchronously notified about the ZCLFrame responses sent back by the ZigBee nodes.

149.31.23.3 public ZCLCommandResponseStream broadcast(int clusterID, ZCLFrame frame, String exportedServicePID)

clusterID The cluster ID.

frame A ZCL Frame.

exportedServicePID the source endpoint of the command request. In targeted situations, the source endpoint is the valid service PID of an exported endpoint.

□ Broadcasts a ZCL frame to the cluster ID of all the nodes of the ZigBee network. The passed `exportedServicePID` allows to force the source endpoint of the message sent to be the endpoint id of the exported ZigBeeEndPoint service having the specified service.pid property.

Returns a response stream instance that collects and allows the caller to be asynchronously notified about the ZCLFrame responses sent back by the ZigBee nodes.

See Also Setting the broadcast radius.

149.31.23.4 public void createGroupService(int groupAddress) throws Exception

groupAddress the address of the group to create.

□ Creates a ZigBeeGroup service that has not yet been discovered by the ZigBee Base Driver or that does not exist on the ZigBee network yet.

Throws Exception– when a ZigBeeGroup service with the same groupAddress already exists.

149.31.23.5 public short getBroadcastRadius()

□ Returns the current broadcast radius value.

Returns the current broadcast radius value.

149.31.23.6 public int getChannel() throws Exception

□ Returns the current network channel.

Returns the current network channel.

Throws Exception– Any exception related to the communication with the chip.

149.31.23.7 public int getChannelMask() throws Exception

□ Returns the currently configured channel mask.

Returns the currently configured channel mask.

Throws Exception– Any exception related to the communication with the chip.

149.31.23.8 public long getCommunicationTimeout()

□ Returns the current value set for the communication timeout.

Returns the current value set for the communication timeout expressed in milliseconds.

149.31.23.9 public String getPreconfiguredLinkKey() throws Exception

□ Returns the current preconfigured link key.

Returns the current preconfigured link key.

Throws Exception– Any exception related to the communication with the chip.

149.31.23.10 public int getSecurityLevel() throws Exception

□ Returns the network security level.

Returns the network security level, that is, 0 if security is disabled, an int code if enabled (see "Table 4.38 Security Levels Available to the NWK, and APS Layers" of the ZigBee specification").

Throws Exception– Any exception related to the communication with the chip.

149.31.23.11 public boolean isStarted()

□ Checks the host's start/stop state.

Returns true if the host is started.

149.31.23.12 public void permitJoin(short duration) throws Exception

duration The time during which associations are permitted.

□ Indicates if a ZigBee device can join the network.

Broadcasts a Mgmt_Permit_req to all routers and the coordinator. If the duration argument is not equal to zero or 0xFF, the argument is a number of seconds and joining is permitted until it counts down to zero, after which time, joining is not permitted. If the duration is set to zero, joining is not permitted. If set to 0xFF, joining is permitted indefinitely or until another Mgmt_Permit_Joining_req is received by the coordinator.

As described in "Table 2.133 Fields of the Mgmt_Permit_Joining_rsp Command" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a permitJoin request can have the following status: ZDPException.SUCCESS, ZDPException.INV_REQUESTTYPE, ZDPException.NOT_AUTHORIZED or any status code returned from the NLMEPERMITJOINING.confirm primitive.

Throws Exception– Any exception related to the communication with the chip.

149.31.23.13 public Promise<Boolean> refreshNetwork() throws Exception

□ Forces a new network scan. It checks that the ZigBeeNode services are still representing an available node on the network. It also updates the whole representation of all nodes (endpoints, clusters, descriptors, attributes).

Returns A promise representing the completion of this asynchronous call. In case of success the promise will resolve with Boolean.TRUE otherwise the promise is failed with an exception.

Throws Exception– Any exception related to the communication with the chip.

149.31.23.14 public void setBroadcastRadius(short broadcastRadius)

broadcastRadius - is the number of routers that the messages are allowed to cross. Radius value is in the range from 0 to 0xff.

□ Sets the broadcast radius value. By default the ZigBeeHost must use UNLIMITED_BROADCAST_RADIUS as default value for the broadcast.

Throws IllegalArgumentException– if set with a value out of the expected range.

IllegalStateException– if set when the ZigBeeHost is "running".

149.31.23.15 public void setChannelMask(int mask) throws IOException

mask A value representing the channel mask.

□ Sets a new configured channel mask.

As described in "Table 2.13 APSME-SET.confirm Parameters" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, a set request can have the following status: APSException.SUCCESS, APSException.INVALID_PARAMETER or APSException.UNSUPPORTED_ATTRIBUTE.

Throws IllegalStateException– If the host is already started.

IOException– for serial communication exception.

149.31.23.16 public void setCommunicationTimeout(long timeout)

timeout the number of milliseconds before firing a timeout exception.

□ Sets the timeout for the communication sent through this device.

149.31.23.17 public void setExtendedPanId(BigInteger extendedPanId)

extendedPanId The network Extended PAN identifier(EPID)

□ Sets the extendedPanId.

Throws IllegalStateException– If the host is already started.

149.31.23.18 public void setLogicalType(short logicalNodeType) throws Exception

logicalNodeType The logical node type.

□ Sets the host logical node type. ZigBee defines three different types of node: ZigBeeNode.COORDINATOR, ZigBeeNode.ROUTER and ZigBeeNode.ZED.

Throws IllegalStateException– If the host is already started.

Exception– Any exception related to the communication with the chip.

149.31.23.19 public void setPanId(int panId)

panId The network Personal Area Network identifier (PAN ID)

□ Sets the panId.

Throws `IllegalArgumentException` – if set with a value out of the expected range [0x0000, 0xffff].

`IllegalStateException` – If the host is already started.

149.31.23.20 public void start() throws Exception

□ Starts the host. If the host is a ZigBeeNode.COORDINATOR, then it can be started with or without ZigBeeNode.PAN_ID and ZigBeeNode.EXTENDED_PAN_ID (that is, if no PAN_ID, and Extended PAN_ID are given, then they will be automatically generated and then added to the service properties).

If the host is a ZigBeeNode.ROUTER, or a ZigBeeNode.ZED, then the host may start without a registered ZigBeeNode.PAN_ID property; the property will be set when the host will find and join a ZigBee network.

The host status must be persistent, that is, if the host was started, then the host must starts again when the bundle restarts. In addition, the values of channel, pan id, extended pan id, and host PID must remain the same.

Throws `Exception` – Any exception related to the communication with the chip.

149.31.23.21 public void stop() throws Exception

□ Stops the host.

Throws `Exception` – Any exception related to the communication with the chip.

149.31.23.22 public void updateNetworkChannel(byte channel) throws IOException

channel The network channel.

□ Updates the network channel. 802.15.4 and ZigBee divide the 2.4GHz band into 16 channels, numbered from 11 to 26.

As described in "Table 2.4.3.3.9 Mgmt_NWK_Update_req" of the ZigBee specification 1_053474r17ZB_TSC-ZigBee-Specification.pdf, this request is sent as broadcast by the network manager with a ScanDuration to be set with the channel parameter.

Throws `IllegalStateException` – If the host is started, or the host is not a network manager.

`IOException` – for serial communication exception.

149.31.24 public interface ZigBeeLinkQuality

This interface represents an entry of the NeighborTableList.

See Table 2.126 NeighborTableList Record Format in ZIGBEE SPECIFICATION: 1_053474r17ZB_TSC-ZigBee-Specification.pdf.

No Implement Consumers of this API must not implement this interface

149.31.24.1 public static final int CHILD_NEIGHBOR = 241

Constant value representing a child relationship between current ZigBeeNode and the neighbor.

149.31.24.2 public static final int OTHERS_NEIGHBOR = 243

Constant value representing a others relationship between current ZigBeeNode and the neighbor.

149.31.24.3 public static final int PARENT_NEIGHBOR = 240

Constant value representing a parent relationship between current ZigBeeNode and the neighbor.

149.31.24.4 public static final int PREVIOUS_CHILD_NEIGHBOR = 244

Constant value representing a previous child relationship between current ZigBeeNode and the neighbor.

149.31.24.5 public static final int SIBLING_NEIGHBOR = 242

Constant value representing a sibling relationship between current ZigBeeNode and the neighbor.

149.31.24.6 public int getDepth()

□ Returns the depth field of the NeighborTableList Record Format.

Returns the tree-depth of device.

149.31.24.7 public int getLQI()

□ Returns the Link Quality Indicator. See the LQI field of the NeighborTableList Record Format.

Returns the Link Quality Indicator estimated by ZigBeeNode returning this for communicating with ZigBeeNode identified by the getNeighbor().

149.31.24.8 public String getNeighbor()

□ Returns the Service.PID referring to the ZigBeeNode representing a neighbor.

Returns the Service.PID referring to the ZigBeeNode representing a neighbor.

149.31.24.9 public int getRelationship()

□ Returns the relationship with the neighbor. See the Relationship field of the NeighborTableList Record Format.

Returns the relationship between ZigBeeNode returning this LQI and the ZigBeeNode identified by the getNeighbor().

149.31.25 public interface ZigBeeNode

This interface represents a ZigBee node, means a physical device that can communicate using the ZigBee protocol.

Each physical device may contain up 240 logical devices which are represented by the ZigBeeEndpoint class.

Each logical device is identified by an *EndPoint* address, but shares:

- either the *64-bit 802.15.4 IEEE Address*
- or the *16-bit ZigBee Network Address.*

No Implement Consumers of this API must not implement this interface

149.31.25.1 public static final short COORDINATOR = 2

Constant value used as logical type value when the ZigBee device is a Coordinator.

149.31.25.2 public static final String EXTENDED_PAN_ID = "zigbee.node.extended.pan.id"

Key of String containing the device node network extended PAN ID. If the device type is "Coordinator", the extended pan id may be available only after the network is started. It means that internally the ZigBeeHost interface must update the service properties.

This property is of type BigInteger

149.31.25.3 public static final String IEEE_ADDRESS = "zigbee.node.ieee.address"

Property key for the mandatory node IEEE Address representing node MAC address. MAC Address is a 12-digit(48-bit) or 16-digit(64-bit) hexadecimal numbers.

149.31.25.4 public static final String LOGICAL_TYPE = "zigbee.node.description.node.type"

Property name for the device logical type. The property value is of type Short.

149.31.25.5 public static final String MANUFACTURER_CODE = "zigbee.node.description.manufacturer.code"

Property name for a manufacturer code that is allocated by the ZigBee Alliance, relating the manufacturer to the device. The property is of type Integer.

149.31.25.6 public static final String PAN_ID = "zigbee.node.pan.id"

Property containing the ZigBee network PAN ID. The property is of type Integer.

149.31.25.7 public static final String POWER_SOURCE = "zigbee.node.power.source"

ZigBee power source, that is, 3rd bit of "MAC Capabilities" in Node Descriptor. Set to `true` if the current power source is mains power, set to `false`, otherwise.

This property is of type Boolean.

149.31.25.8 public static final String RECEIVER_ON_WHEN_IDLE = "zigbee.node.receiver.on.when.idle"

ZigBee receiver on when idle, that is, 4th bit of "MAC Capabilities" in Node Descriptor. Set to `true` if the device does not disable its receiver to conserve power during idle periods, set to `false` otherwise.

This property is of type Boolean.

149.31.25.9 public static final short ROUTER = 3

Constant value used as logical type value when the ZigBee device is a Router.

149.31.25.10 public static final short ZED = 1

Constant value used as logical type value when the ZigBee device is an End Device.

149.31.25.11 public ZCLCommandResponseStream broadcast(int clusterID, ZCLFrame frame)

clusterID the cluster ID the broadcast message is directed.

frame a ZCL Frame that contains the command that have to be broadcast to the specific cluster of all the endpoints running on the node.

□ Broadcasts a given ZCL Frame to cluster `clusterID` on all the ZigBeeEndpoint that are running on this node (endpoint broadcasting).

Returns a response stream instance that collects and allows a client to be asynchronously notified about the ZCLFrame responses sent back by the ZigBee nodes.

149.31.25.12 public ZCLCommandResponseStream broadcast(int clusterID, ZCLFrame frame, String exportedServicePID)

clusterID the cluster ID the broadcast message is directed.

frame a ZCL Frame that contains the command that have to be broadcast to the specific cluster of all the endpoints running on the node.

exportedServicePID the source endpoint of the command request. In targeted situations, the source endpoint is the valid service PID of an exported endpoint.

□ Broadcasts a given ZCL Frame to cluster `clusterID` on all the ZigBeeEndpoint that are running on this node (endpoint broadcasting). The source endpoint of the APS message sent, is set to the endpoint identifier of the `exportedServicePID` service.

Returns a response stream instance that collects and allows the caller to be asynchronously notified about the ZCLFrame responses sent back by the ZigBee nodes.

149.31.25.13 public Promise<ZigBeeComplexDescriptor> getComplexDescriptor()

□ Retrieves the ZigBee node Complex Descriptor.

As described in *Table 2.92 Fields of the Complex_Desc_rsp Command* of the ZigBee specification, a *Complex_Desc_rsp* command can return with the following status codes:

- ZDPException.SUCCESS
- ZDPException.DEVICE_NOT_FOUND
- ZDPException.INV_REQUESTTYPE
- ZDPException.NO_DESCRIPTOR

Returns A promise representing the completion of this asynchronous call. It will be used in order to return the complex descriptor ZigBeeComplexDescriptor. If the ZDP *Complex_Desc_rsp* do not return success, the promise must fail with a ZDPException exception with the correct status code.

149.31.25.14 public ZigBeeEndpoint[] getEndpoints()

□ Returns the array of the endpoints hosted by this node.

Returns the array of the endpoints hosted by this node, returns an empty array if this node does not host any endpoint.

149.31.25.15 public BigInteger getExtendedPanId()

□ Returns the network Extended PAN identifier (EPID).

Returns the network Extended PAN identifier (EPID).

149.31.25.16 public String getHostPid()

□ Returns the OSGi service PID of the ZigBee Host that is on the network of this node.

Returns the OSGi service PID of the ZigBee Host that is on the network of this node.

149.31.25.17 public BigInteger getIEEEAddress()

□ Returns the ZigBee device node IEEE Address of this node.

Returns the ZigBee device node IEEE Address of this node.

149.31.25.18 public Promise<Map<String, ZigBeeLinkQuality>> getLinksQuality()

□ Retrieves the link quality information to the neighbor nodes.

An implementation of this method may use the *Mgmt_Lqi_req* and *Mgmt_Lqi_rsp* messages to retrieve the Link Quality table (also known as NeighborTableList in the ZigBee Specification).

The method limit the Link Quality table to the ZigBeeNode service discovered.

In case of failure, the target device may report error code ZDPException.NOT_SUPPORTED.

Returns A promise representing the completion of this asynchronous call. It will be resolved with the result of this operation. In case of success the resolved value will be a Map containing the Service.PID as String key of the ZigBeeNode service and the value the ZigBeeLinkQuality for that node. In case of errors the promise must fail with the correct ZDPException.

149.31.25.19 public int getNetworkAddress()

□ Returns the current network address (alias short-address) of this node.

Returns the current network address of this node.

149.31.25.20 public Promise<ZigBeeNodeDescriptor> getNodeDescriptor()

□ Retrieves the ZigBee node Node Descriptor. As described in *Table 2.91 Fields of the Node_Desc_rsp Command* of the ZigBee specification, a *Node_Desc_rsp* command can return with the following status codes:

- ZDPException.SUCCESS
- ZDPException.DEVICE_NOT_FOUND
- ZDPException.INV_REQUESTTYPE
- ZDPException.NO_DESCRIPTOR

Returns A promise representing the completion of this asynchronous call. It will be used in order to return the node descriptor ZigBeeNodeDescriptor. If the ZDP *Node_Desc_rsp* do not return success, the promise must fail with a ZDPException exception with the correct status code.

149.31.25.21 public int getPanId()

□ Returns the network Personal Area Network identifier (PAN ID).

Returns the network Personal Area Network identifier (PAN ID).

149.31.25.22 public Promise<ZigBeePowerDescriptor> getPowerDescriptor()

□ Retrieves the ZigBee node Power Descriptor. As described in *Table 2.92 Fields of the Power_Desc_rsp Command* of the ZigBee specification, a *Power_Desc_rsp* command can return with the following status codes:

- ZDPException.SUCCESS
- ZDPException.DEVICE_NOT_FOUND
- ZDPException.INV_REQUESTTYPE
- ZDPException.NO_DESCRIPTOR

Returns A promise representing the completion of this asynchronous call. It will be used in order to return the node power descriptor ZigBeePowerDescriptor. If the ZDP *Power_Desc_rsp* do not return success, the promise must fail with a ZDPException exception with the correct status code.

149.31.25.23 public Promise<Map<String, ZigBeeRoute>> getRoutingTable()

□ Retrieves the routing table information of the node. This routing table is also known as RoutingTableList in the ZigBee Specification.

An implementation of this method may use the *Mgmt_Rtg_req* ZDP command to retrieve the Routing Table .

The target device may report a status code ZDPException.NOT_SUPPORTED in case of failure.

Returns A promise representing the completion of this asynchronous call. In case of success, the resolved value will be a Map containing the Service.PID as String key of the ZigBeeNode service and the value the ZigBeeRoute for that node. In case of failure a ZDPException exception with the correct status code must be used to fail the promise.

149.31.25.24 public Promise<String> getUserDescription()

□ Returns the user description of this node. As described in *Table 2.97 Fields of the User_Desc_rsp Command* of the ZigBee specification, a User_Desc_rsp may return the following status:

- ZDPException.SUCCESS
- ZDPException.NOT_SUPPORTED
- ZDPException.DEVICE_NOT_FOUND
- ZDPException.INV_REQUESTTYPE
- ZDPException.NO_DESCRIPTOR

Returns A promise representing the completion of this asynchronous call. It will be used in order to return the node user description (String). In case of errors the promise will fail with a ZDPException exception containing the response status code value.

149.31.25.25 public Promise<ZDPFrame> invoke(int clusterIdReq, int expectedClusterIdRsp, ZDPFrame message)

clusterIdReq the cluster Id of the ZDPFrame that will be sent to the device.

expectedClusterIdRsp the expected cluster Id of the response to the ZDPFrame sent.

message the ZDPFrame containing the message.

□ Sends the ZDPFrame to this ZigBeeNode with the specified cluster id. This method expects a specific cluster in the response to the request.

Returns A promise representing the completion of this asynchronous call. In case of success the promise resolves with the response ZDPFrame.

149.31.25.26 public Promise<ZDPFrame> invoke(int clusterIdReq, ZDPFrame message)

clusterIdReq the cluster Id of the ZDPFrame that will be sent to the device.

message the ZDPFrame containing the message.

□ Sends the ZDPFrame to this ZigBeeNode with the specified cluster id. This method expects a specific cluster in the response to the request. This method considers that the 0x8000 + clusterIdReq is the clusterId expected from messaged received for the message sent by this request.

Returns A promise representing the completion of this asynchronous call. In case of success the promise resolves with the response ZDPFrame.

149.31.25.27 public Promise<Void> leave()

□ Requests this node to leave the ZigBee network.

As described in *Table 2.131 Fields of the Mgmt_Leave_rsp Command* of the ZigBee specification, a Mgmt_Leave_rsp ZDP command may return the following status values:

- ZDPException.SUCCESS
- ZDPException.NOT_SUPPORTED
- ZDPException.NOT_AUTHORIZED
- any status code returned from the NLMELEAVE.confirm primitive

Returns A promise representing the completion of this asynchronous call. In case of success, the promise is resolved with a null value, otherwise with the correct ZDPException exception.

149.31.25.28 public Promise<Void> leave(boolean rejoin, boolean removeChildren)

rejoin true if the device being asked to leave from the current parent is requested to rejoin the network. Otherwise, false.

removeChildren true if the device being asked to leave the network is also being asked to remove its child devices, if any. Otherwise, false.

□ Requests the device to leave the network.

As described in *Table 2.131 Fields of the Mgmt_Leave_rsp Command* of the ZigBee specification, a Mgmt_Leave_rsp command could return the following status values:

- ZDPException.SUCCESS
- ZDPException.NOT_SUPPORTED
- ZDPException.NOT_AUTHORIZED
- any status code returned from the NLMELEAVE.confirm primitive

Returns A promise representing the completion of this asynchronous call. In case of success, the ZigBeeNode service must be unregistered, first and then the promise may be resolved with a null value, otherwise with the correct ZDPException exception.

149.31.25.29 public Promise<Void> setUserDescription(String userDescription)

userDescription the user description.

□ Sets the user description of this node. As described in *Table 2.137 ZDP Enumerations Description* of the ZigBee specification, a Set_User_Desc_rsp request may return the following status:

- ZDPException.SUCCESS
- ZDPException.DEVICE_NOT_FOUND
- ZDPException.INV_REQUESTTYPE
- ZDPException.NO_DESCRIPTOR

Returns A promise representing the completion of this asynchronous call. In case of success the promise returns a null value. In case of errors the promise must fail with a ZDPException exception containing the response status code value.

149.31.26 public interface ZigBeeRoute

This interface represents an entry of the RoutingTableList

See Table 2.128 RoutingTableList Record Format in ZIGBEE SPECIFICATION: 1_053474r17ZB_TSC-ZigBee-Specification.pdf.

No Implement Consumers of this API must not implement this interface

149.31.26.1 public static final int ACTIVE = 240

Constant value representing an active route.

149.31.26.2 public static final int DISCOVERY_FAILED = 242

Constant value representing a failed route discovery.

149.31.26.3 public static final int DISCOVERY_UNDERWAY = 241

Constant value representing a route that is under discovery.

149.31.26.4 public static final int INACTIVE = 243

Constant value representing an inactive route.

149.31.26.5 public static final int VALIDATION_UNDERWAY = 244

Constant value representing a route which is under validation.

149.31.26.6 public String getDestination()

□ Returns the service PID of the ZigBeeNode as destination of this route entry.

Returns the service PID of the ZigBeeNode as destination of this route entry.

149.31.26.7 public String getNextHop()

□ Returns the service PID of the ZigBeeNode to send the data for reaching the destination.

Returns the service PID of the ZigBeeNode to send the data for reaching the destination.

149.31.26.8 public int getStatus()

□ Returns the status of this route.

Returns the status of this route (or routing link) as defined by ZigBee Specification: ACTIVE, DISCOVERY_UNDERWAY, DISCOVERY_FAILED, INACTIVE, VALIDATION_UNDERWAY.

149.32 org.osgi.service.zigbee.descriptions

Device Service Specification for ZigBee Technology Descriptions.

This package contains the interfaces for descriptions. The latter may be used to embed meta information about the ZigBee devices, and in other words a meta description of each device type present in a ZCL profile, or even custom devices.

It is not mandatory to provide this meta model for being able to interact with a specific device, but the presence of this meta model would make much easier to implement, for example user interfaces.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.zigbee.descriptions; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.zigbee.descriptions; version="[1.0,1.1)"
```

149.32.1 Summary

- `ZCLAttributeDescription` - This interface represents a ZCLAttributeDescription.
- `ZCLClusterDescription` - This interface represents a ZCL Cluster description.
- `ZCLCommandDescription` - This interface represents a ZCLCommandDescription.
- `ZCLDataTypeDescription` - This interface is used for representing any of the ZigBee Data Types defined in the ZCL.
- `ZCLGlobalClusterDescription` - This interface represents Cluster global description.
- `ZCLParameterDescription` - This interface represents a ZigBee parameter description.
- `ZCLSimpleTypeDescription` - This interface is used for representing any of the simple ZigBee Data Types defined in the ZCL.
- `ZigBeeDeviceDescription` - This interface represents a ZigBee device description.
- `ZigBeeDeviceDescriptionSet` - This interface represents a ZigBee Device description Set.

149.32.2 public interface ZCLAttributeDescription extends ZCLAttributeInfo

This interface represents a ZCLAttributeDescription.

149.32.2.1 public Object getDefaultValue()

□ Returns the attribute default value.

Returns the attribute default value.

149.32.2.2 public String getName()

□ Returns the attribute name.

Returns the attribute name.

149.32.2.3 public String getShortDescription()

□ Returns the attribute functional description.

Returns the attribute functional description.

149.32.2.4 public boolean isMandatory()

□ Checks if this attribute is mandatory.

Returns true, if and only if the attribute is mandatory.

149.32.2.5 public boolean isPartOfAScene()

□ Checks if this attribute is part of a scene.

Returns true if the attribute is part of a scene (cluster), false otherwise.

149.32.2.6 public boolean isReadOnly()

□ Checks if this attribute is read-only.

Returns true if the attribute is read only, false otherwise (that is, if the attribute is read/write or optionally writable (R*W)).

149.32.2.7 public boolean isReportable()

□ Checks if this attribute is reportable.

Returns true if and only if the attribute support subscription.

149.32.3 public interface ZCLClusterDescription

This interface represents a ZCL Cluster description.

149.32.3.1 public ZCLAttributeDescription[] getAttributeDescriptions()

□ Returns an array of the attribute descriptions.

Returns an array of the attribute descriptions.

149.32.3.2 public ZCLCommandDescription[] getGeneratedCommandDescriptions()

□ Returns an array of the generated command descriptions.

Returns an array of the generated command descriptions.

149.32.3.3 public ZCLGlobalClusterDescription getGlobalClusterDescription()

□ Returns an array of the command descriptions.

Returns an array of the command descriptions.

149.32.3.4 public int getId()

Returns the cluster identifier.

149.32.3.5 public ZCLCommandDescription[] getReceivedCommandDescriptions()

□ Returns an array of the received command description.

Returns an array of the received command description.

149.32.4 public interface ZCLCommandDescription

This interface represents a ZCLCommandDescription.

149.32.4.1 public short getId()

□ Returns the command identifier.

Returns the command identifier.

149.32.4.2 public int getManufacturerCode()

□ Returns the manufacturer code. Default value is: -1 (no code).

Returns the manufacturer code.

149.32.4.3 public String getName()

□ Returns the command name.

Returns the command name.

149.32.4.4 public ZCLParameterDescription[] getParameterDescriptions()

□ Returns an array of the parameter descriptions.

Returns an array of the parameter descriptions.

149.32.4.5 public String getShortDescription()

□ Returns the command functional description.

Returns the command functional description.

149.32.4.6 public boolean isClientServerDirection()

□ Checks if this is a server-side command (that is going from the client to server direction).

Returns the isClientServerDirection value.

149.32.4.7 public boolean isClusterSpecificCommand()

Returns the isClusterSpecificCommand value.

149.32.4.8 public boolean isMandatory()

□ Checks if this command it mandatory.

Returns true, if and only if the command is mandatory.

149.32.4.9 public boolean isManufacturerSpecific()

□ Checks if the command is manufacturer specific.

Returns `true` if end only if getManufacturerCode() is not. -1.

149.32.5 public interface ZCLDataTypeDescription

This interface is used for representing any of the ZigBee Data Types defined in the ZCL. Each of these data types has a set of associated information that this interface definition permits to retrieve using the specific methods.

- The data type identifier
- The data type name
- The data type is analog or digital
- The Java class used to represent the data type.

No Implement Consumers of this API must not implement this interface

149.32.5.1 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.32.5.2 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.32.5.3 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.32.5.4 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.32.6 public interface ZCLGlobalClusterDescription

This interface represents Cluster global description.

149.32.6.1 public ZCLClusterDescription getClientClusterDescription()

□ Returns the client cluster description.

Returns the client cluster description.

149.32.6.2 public String getClusterDescription()

□ Returns the cluster functional description.

Returns the cluster functional description.

149.32.6.3 public String getClusterFunctionalDomain()

□ Returns the cluster functional domain.

Returns the cluster functional domain.

149.32.6.4 public int getClusterId()

□ Returns the cluster identifier.

Returns the cluster identifier.

149.32.6.5 public String getClusterName()

□ Returns the cluster name.

Returns the cluster name.

149.32.6.6 public ZCLClusterDescription getServerClusterDescription()

□ Returns the server cluster description.

Returns the server cluster description.

149.32.7 public interface ZCLParameterDescription

This interface represents a ZigBee parameter description.

149.32.7.1 public ZCLDataTypeDescription getDataTypeDescription()

□ Returns the parameter data type.

Returns the parameter data type.

149.32.8 public interface ZCLSimpleTypeDescription extends ZCLDataTypeDescription

This interface is used for representing any of the simple ZigBee Data Types defined in the ZCL.

The interface extends the ZCLDataTypeDescription by providing serialize and deserialize methods to marshal and unmarshal the data into the ZigBeeDataInput and from ZigBeeDataOutput streams.

Related documentation: [1] ZigBee Cluster Library specification, Document 075123r04ZB, May 29, 2012.

No Implement Consumers of this API must not implement this interface

149.32.8.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws NullPointerException– If ZigBeeDataInput parameter is null.

IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.32.8.2 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

NullPointerException– If ZigBeeDataOutput parameter is null.

IllegalArgumentException– If the passed value parameter does not belong to the expected class or its value exceeds the possible values allowed (range or length).

149.32.9 public interface ZigBeeDeviceDescription

This interface represents a ZigBee device description.

149.32.9.1 public ZCLClusterDescription[] getClientClustersDescriptions()

□ Returns an array of client cluster descriptions.

Returns an array of client cluster descriptions.

149.32.9.2 public int getId()

□ Returns the device identifier.

Returns the device identifier.

149.32.9.3 public String getName()

□ Returns the device name.

Returns the device name.

149.32.9.4 public int getProfileId()

□ Returns the profile identifier.

Returns the profile identifier.

149.32.9.5 public ZCLClusterDescription[] getServerClustersDescriptions()

□ Returns an array of server cluster descriptions.

Returns an array of server cluster descriptions.

149.32.9.6 public Integer getVersion()

□ Returns the device version.

Returns the device version.

149.32.10 public interface ZigBeeDeviceDescriptionSet

This interface represents a ZigBee Device description Set. A Set is registered as an OSGi Service that provides method to retrieve endpoint descriptions. In addition to the ZigBeeDeviceDescriptionSet's (OSGi service) properties; ZigBeeDeviceDescriptionSet is also expected to be registered as an OSGi service with the following ZigBeeEndpoint.PROFILE_ID, and ZigBeeNode.MANUFACTURER_CODE properties.

149.32.10.1 public static final String DEVICES = "zigbee.profile.devices"

Property key for a comma separated list of devices identifiers supported by the set. This property is **mandatory**.

149.32.10.2 public static final String PROFILE_NAME = "zigbee.profile.name"

Property key for a profile name. This property is **mandatory**.

149.32.10.3 public static final String VERSION = "zigbee.profile.version"

Property key for a version of the application profile. The format is 'major.minor' with major and minor being integers. This property is **mandatory**.

149.32.10.4 public ZigBeeDeviceDescription getDeviceSpecification(int deviceId, short version)

deviceId Identifier of the device.

version The version of the application profile.

□ Returns the description of a device identified by its identifier and its version.

Returns The associated device description.

149.33 org.osgi.service.zigbee.descriptors

Device Service Specification for ZigBee Technology Descriptors.

This package contains the interfaces representing the ZigBee descriptors and the fields defined inside some of them.

An interface for modeling the ZigBee User Descriptor is missing because this descriptor has only one field (the UserDescription). Therefore this field can be read and written using respectively the org.osgi.service.zigbee.ZigBeeNode.getUserDescription() and the org.osgi.service.zigbee.ZigBeeNode.setUserDescription(String) methods.

The org.osgi.service.zigbee.descriptors.ZigBeeNodeDescriptor, org.osgi.service.zigbee.descriptors.ZigBeePowerDescriptor and the org.osgi.service.zigbee.descriptors.ZigBeeComplexDescriptor are read using the appropriate methods in the org.osgi.service.zigbee.ZigBeeNode interface, whereas the ZigBeeSimpleDescriptor can be read using the appropriate method of the org.osgi.service.zigbee.ZigBeeEndpoint services registered in the framework.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.zigbee.descriptors; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.zigbee.descriptors; version="[1.0,1.1)"
```

149.33.1 Summary

- `ZigBeeComplexDescriptor` - This interface represents a Complex Descriptor as described in the ZigBee Specification.
- `ZigBeeFrequencyBand` - This interface represents a the frequency band field.
- `ZigBeeMacCapabiliyFlags` - This interface represents the Node Descriptor MAC Capability Flags as described in the ZigBee Specification.
- `ZigBeeNodeDescriptor` - This interface represents a Node Descriptor as described in the ZigBee Specification.
- `ZigBeePowerDescriptor` - This interface represents a power descriptor as described in the ZigBee Specification.
- `ZigBeeServerMask` - Represents the ZigBee Server Mask field of the ZigBee Node Descriptor.
- `ZigBeeSimpleDescriptor` - This interface represents a simple descriptor as described in the ZigBee Specification.

149.33.2 public interface ZigBeeComplexDescriptor

This interface represents a Complex Descriptor as described in the ZigBee Specification.

The Complex Descriptor contains extended information for each of the device descriptions contained in the node. The use of the Complex Descriptor is optional.

No Implement Consumers of this API must not implement this interface

149.33.2.1 public String getCharacterSetIdentifier()

□ Returns the encoding used by characters in the character set.

Returns the encoding used by characters in the character set.

149.33.2.2 public String getDeviceURL()

□ Returns the Device URL.

Returns the Device URL.

149.33.2.3 public byte[] getIcon()

□ Returns the icon field.

Returns the icon field.

149.33.2.4 public String getIconURL()

□ Returns the icon URL.

Returns the icon URL.

149.33.2.5 public String getLanguageCode()

□ Returns the language code used for character strings.

Returns the language code used for character strings.

149.33.2.6 public String getManufacturerName()

□ Returns the manufacturer name.

Returns the manufacturer name.

149.33.2.7 public String getModelName()

□ Returns the model name.

Returns the model name.

149.33.2.8 public String getSerialNumber()

□ Returns the serial number.

Returns the serial number.

149.33.3 public interface ZigBeeFrequencyBand

This interface represents a the frequency band field.

No Implement Consumers of this API must not implement this interface

149.33.3.1 public boolean is2400()

□ Checks if the radio band is 2.4GHz.

Returns `true` if and only if the radio is operating in the frequency band 2400MHz to 2483MHz.

149.33.3.2 public boolean is868()

□ Checks if the radio band is 868MHz.

Returns `true` if and only if the radio is operating in the frequency band 868 to 868.6 MHz.

149.33.3.3 public boolean is915()

□ Checks if the radio band is 900MHz.

Returns `true` if and only if the radio is operating in the frequency band 908MHz to 928MHz.

149.33.4 public interface ZigBeeMacCapabiliyFlags

This interface represents the Node Descriptor MAC Capability Flags as described in the ZigBee Specification.

No Implement Consumers of this API must not implement this interface

149.33.4.1 public boolean isAddressAllocate()

□ Checks if the device is address allocate.

Returns true if the device is address allocate or false otherwise.

149.33.4.2 public boolean isAlternatePANCoordinator()

□ Checks if this node is capable of becoming PAN coordinator.

Returns true if this node is capable of becoming PAN coordinator or false otherwise.

149.33.4.3 public boolean isFullFunctionDevice()

□ Checks if this node a Full Function Device (FFD).

Returns true if this node a Full Function Device (FFD), false otherwise (it is a Reduced Function Device, RFD).

149.33.4.4 public boolean isMainsPower()

□ Checks if the current power source is mains power.

Returns true if the current power source is mains power or false otherwise.

149.33.4.5 public boolean isReceiverOnWhenIdle()

□ Checks if the device does not disable its receiver to conserve power during idle periods.

Returns true if the device does not disable its receiver to conserve power during idle periods or false otherwise.

149.33.4.6 public boolean isSecurityCapable()

□ Checks if the device is capable of sending and receiving secured frames

Returns true if the device is capable of sending and receiving secured frames or false otherwise.

149.33.5 public interface ZigBeeNodeDescriptor

This interface represents a Node Descriptor as described in the ZigBee Specification.

The Node Descriptor contains information about the capabilities of the node.

No Implement Consumers of this API must not implement this interface

149.33.5.1 public ZigBeeFrequencyBand getFrequencyBand()

□ Returns the radio frequency band the node is currently operating on.

Returns returns the information about the radio frequency band the node is currently operating on.

149.33.5.2 public short getLogicalType()

□ Returns the logical type of the described node.

Returns one of: ZigBeeNode.COORDINATOR, ZigBeeNode.ROUTER, ZigBeeNode.ZED.

149.33.5.3 public ZigBeeMacCapabiliyFlags getMacCapabilityFlags()

□ Returns the MAC Capability Flags field information.

Returns the MAC Capability Flags field information.

149.33.5.4 public int getManufacturerCode()

□ Returns the manufacturer code of the described node.

Returns the manufacturer code of the described node.

149.33.5.5 public int getMaxBufferSize()

□ Returns the maximum buffer size of the described node.

Returns the maximum buffer size of the described node.

149.33.5.6 public int getMaxIncomingTransferSize()

□ Returns the maximum incoming transfer size of the described node.

Returns the maximum incoming transfer size of the described node.

149.33.5.7 public int getMaxOutgoingTransferSize()

□ Returns the maximum outgoing transfer size of the described node.

Returns the maximum outgoing transfer size of the described node.

149.33.5.8 **public ZigBeeServerMask getServerMask()**

□ Returns the server mask of the described node.

Returns the server mask of the described node.

149.33.5.9 **public boolean isComplexDescriptorAvailable()**

□ Checks if a complex descriptor is available.

Returns true if a complex descriptor is available or false otherwise.

149.33.5.10 **public boolean isExtendedActiveEndpointListAvailable()**

□ Checks if extended active endpoint list is available.

Returns true if extended active endpoint list is available or false otherwise.

149.33.5.11 **public boolean isExtendedSimpleDescriptorListAvailable()**

□ Checks if extended simple descriptor is available.

Returns true if extended simple descriptor is available or false otherwise.

149.33.5.12 **public boolean isUserDescriptorAvailable()**

□ Checks if a user descriptor is available.

Returns true if a user descriptor is available or false otherwise.

149.33.6 public interface ZigBeePowerDescriptor

This interface represents a power descriptor as described in the ZigBee Specification.

The Power Descriptor gives a dynamic indication of the power status of the node.

No Implement Consumers of this API must not implement this interface

149.33.6.1 **public static final short CRITICAL_LEVEL = 0**

Current power source level: critical.

149.33.6.2 **public static final short FULL_LEVEL = 3**

Current power source level: 100%.

149.33.6.3 **public static final short LOW_LEVEL = 1**

Current power source level: 33%.

149.33.6.4 **public static final short MIDDLE_LEVEL = 2**

Current power source level: 66%.

149.33.6.5 **public short getCurrentPowerMode()**

□ Returns the current power mode.

Returns the current power mode.

149.33.6.6 **public short getCurrentPowerSource()**

□ Returns the current power source field of the Power Descriptor.

Returns the current power source field of the Power Descriptor.

149.33.6.7 **public short getCurrentPowerSourceLevel()**

□ Returns the current power source level.

Returns the current power source level. May be one of CRITICAL_LEVEL, LOW_LEVEL, MIDDLE_LEVEL, FULL_LEVEL.

149.33.6.8 public boolean isConstantMainsPowerAvailable()

□ Checks if constant (mains) power is available.

Returns true if constant (mains) power is available or false otherwise.

149.33.6.9 public boolean isDisposableBattery()

□ Checks if the currently selected power source is the disposable battery.

Returns true if the currently selected power source is the disposable battery.

149.33.6.10 public boolean isDisposableBatteryAvailable()

□ Checks if a disposable battery is available.

Returns true if a disposable battery is available or false otherwise.

149.33.6.11 public boolean isMainsPower()

□ Checks if the currently selected power source is the mains power.

Returns true if the currently selected power source is the mains power.

149.33.6.12 public boolean isOnWhenStimulated()

□ Checks if the receiver is on when the device is simulated.

Returns true if the Current Power Mode field tells that the receiver is on when the device is stimulated by pressing a button, for instance.

149.33.6.13 public boolean isPeriodicallyOn()

□ Checks if the Current Power Mode field is periodically on.

Returns true if the Current Power Mode field is periodically on.

149.33.6.14 public boolean isRechargableBattery()

□ Checks if the currently selected power source is the rechargeable battery.

Returns true if the currently selected power source is the rechargeable battery.

149.33.6.15 public boolean isRechargableBatteryAvailable()

□ Checks if a rechargeable battery is available.

Returns true if a rechargeable battery is available or false otherwise.

149.33.6.16 public boolean isSyncronizedWithOnIdle()

□ Checks if synchronized with the receiver on-when-idle subfield of the node descriptor.

Returns true if the Current Power Mode field is synchronized on idle.

149.33.7 public interface ZigBeeServerMask

Represents the ZigBee Server Mask field of the ZigBee Node Descriptor.

No Implement Consumers of this API must not implement this interface

149.33.7.1 public boolean isBackupBindingTableCache()

□ Checks if the server is a Backup Binding Table Cache.

Returns true if and only if the server is a Backup Binding Table Cache.

149.33.7.2 public boolean isBackupDiscoveryCache()

- Checks if the server is a Backup Discovery Cache.

Returns true if and only if the server is a Backup Discovery Cache.

149.33.7.3 public boolean isBackupTrustCenter()

- Checks if the server is a Backup Trust Center.

Returns true if and only if the server is a Backup Trust Center.

149.33.7.4 public boolean isNetworkManager()

- Checks if the server is a Network Manager.

Returns true if and only if the server is a Network Manager.

149.33.7.5 public boolean isPrimaryBindingTableCache()

- Checks if the server is a Primary Binding Table Cache.

Returns true if and only if the server is a Primary Binding Table Cache.

149.33.7.6 public boolean isPrimaryDiscoveryCache()

- Checks if the server is a Primary Discovery Cache.

Returns true if and only if the server is a Primary Discovery Cache.

149.33.7.7 public boolean isPrimaryTrustCenter()

- Checks if the server is a Primary Trust Center.

Returns true if and only if the server is a Primary Trust Center.

149.33.8 public interface ZigBeeSimpleDescriptor

This interface represents a simple descriptor as described in the ZigBee Specification.

The Simple Descriptor contains information specific to each endpoint present in the node.

149.33.8.1 public int getApplicationDeviceId()

- Returns the application device id as defined per profile.

Returns the application device id as defined per profile.

149.33.8.2 public byte getApplicationDeviceVersion()

- Returns the version of the endpoint application.

Returns the version of the endpoint application.

149.33.8.3 public int getApplicationProfileId()

- Returns the application profile id.

Returns the application profile id.

149.33.8.4 public short getEndpoint()

- Returns the endpoint for which this descriptor is defined.

Returns the endpoint for which this descriptor is defined.

149.33.8.5 public int[] getInputClusters()

- Returns an array of input (server) cluster identifiers.

Returns an array of input (server) cluster identifiers, returns an empty array if does not provides any input (server) clusters.

149.33.8.6 public int[] getOutputClusters()

□ Returns an array of output (client) cluster identifiers.

Returns an array of output (client) cluster identifiers, returns an empty array if does not provides any output (client) clusters.

149.33.8.7 public boolean providesInputCluster(int clusterId)

clusterId the cluster identifier.

□ Checks if this endpoint implements the given cluster id as an input cluster.

Returns true if and only if this endpoint implements the given cluster id as an input cluster.

149.33.8.8 public boolean providesOutputCluster(int clusterId)

clusterId the cluster identifier.

□ Checks if this endpoint implements the given cluster id as an output cluster.

Returns true if and only if this endpoint implements the given cluster id as an output cluster.

149.34 org.osgi.service.zigbee.types

Device Service Specification for ZigBee Technology Data Types.

Utility classes modeling the ZCL data types. Each class provides the static getInstance() method for retrieving a singleton instance of the class itself.

Every class contains methods for getting information about the data type like its ID and name. It is also possible to know if the data type is analog or digital or get the Java class it is mapped in.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.zigbee.types; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.zigbee.types; version="[1.0,1.1)"`

See Also org.osgi.service.zigbee.descriptions.ZCLDataTypeDescription

149.34.1 Summary

- `ZigBeeArray` - A singleton class that represents the 'Array' data type, as it is defined in the ZigBee Cluster Library specification.
- `ZigBeeAttributeID` - A singleton class that represents the 'Attribute ID' data type, as it is defined in the ZigBee Cluster Library specification.
- `ZigBeeBACnet` - A singleton class that represents the 'Unsigned Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- `ZigBeeBag` - A singleton class that represents the 'Bag' data type, as it is defined in the ZigBee Cluster Library specification.
- `ZigBeeBitmap16` - A singleton class that represents the 'Bitmap 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

- ZigBeeBitmap24 - A singleton class that represents the 'Bitmap 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap32 - A singleton class that represents the 'Bitmap 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap40 - A singleton class that represents the 'Bitmap 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap48 - A singleton class that represents the 'Bitmap 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap56 - A singleton class that represents the 'Bitmap 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap64 - A singleton class that represents the 'Bitmap 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBitmap8 - A singleton class that represents the 'Bitmap 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeBoolean - A singleton class that represents the 'Boolean' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeCharacterString - A singleton class that represents the 'Character String' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeClusterID - A singleton class that represents the 'Cluster ID' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeDate - A singleton class that represents the 'Date' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeEnumeration16 - A singleton class that represents the 'Enumeration 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeEnumeration8 - A singleton class that represents the 'Enumeration 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeFloatingDouble - A singleton class that represents the 'Floating Double' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeFloatingSemi - A singleton class that represents the 'Floating Semi' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeFloatingSingle - A singleton class that represents the 'Floating Single' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData16 - A singleton class that represents the 'General Data 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData24 - A singleton class that represents the 'General Data 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData32 - A singleton class that represents the 'General Data 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData40 - A singleton class that represents the 'General Data 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData48 - A singleton class that represents the 'General Data 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData56 - A singleton class that represents the 'General Data 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData64 - A singleton class that represents the 'General Data 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeGeneralData8 - A singleton class that represents the 'General Data 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeIEEE_ADDRESS - A singleton class that represents the 'IEEE ADDRESS' data type, as it is defined in the ZigBee Cluster Library specification.

- ZigBeeLongCharacterString - A singleton class that represents the 'Long Character String' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeLongOctetString - A singleton class that represents the 'Long Octet String' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeOctetString - A singleton class that represents the 'Octet String' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSecurityKey128 - A singleton class that represents the 'Security Key 128' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSet - A singleton class that represents the 'Set' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger16 - A singleton class that represents the 'Signed Integer 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger24 - A singleton class that represents the 'Signed Integer 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger32 - A singleton class that represents the 'Signed Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger40 - A singleton class that represents the 'Signed Integer 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger48 - A singleton class that represents the 'Signed Integer 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger56 - A singleton class that represents the 'Signed Integer 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger64 - A singleton class that represents the 'Signed Integer 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeSignedInteger8 - A singleton class that represents the 'Signed Integer 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeStructure - A singleton class that represents the 'Structure' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeTimeOfDay - A singleton class that represents the 'Time Of Day' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger16 - A singleton class that represents the 'Unsigned Integer 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger24 - A singleton class that represents the 'Unsigned Integer 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger32 - A singleton class that represents the 'Unsigned Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger40 - A singleton class that represents the 'Unsigned Integer 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger48 - A singleton class that represents the 'Unsigned Integer 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger56 - A singleton class that represents the 'Unsigned Integer 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger64 - A singleton class that represents the 'Unsigned Integer 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUnsignedInteger8 - A singleton class that represents the 'Unsigned Integer 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.
- ZigBeeUTCTime - A singleton class that represents the 'UTC Time' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.2 public class ZigBeeArray implements ZCLDataTypeDescription

A singleton class that represents the 'Array' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.2.1 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.2.2 public static ZigBeeArray getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.2.3 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.2.4 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.2.5 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.3 public class ZigBeeAttributeID implements ZCLSimpleTypeDescription

A singleton class that represents the 'Attribute ID' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.3.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns `null` if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws `IOException`– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.3.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.3.3 public static ZigBeeAttributeID getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.3.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.3.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.3.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.3.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.4 public class ZigBeeBACnet implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.4.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.4.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.4.3 public static ZigBeeBACnet getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.4.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.4.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.4.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.4.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be `null`. If `null` a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a `null` value as the request to serialize the so called *Invalid Value*.

Throws `IOException`– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.5 public class ZigBeeBag implements ZCLDataTypeDescription

A singleton class that represents the 'Bag' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.5.1 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.5.2 public static ZigBeeBag getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.5.3 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.5.4 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.5.5 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.6 public class ZigBeeBitmap16 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.6.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns `null` if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws `IOException`– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.6.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.6.3 public static ZigBeeBitmap16 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.6.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.6.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.6.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.6.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.7 public class ZigBeeBitmap24 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.7.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.7.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.7.3 public static ZigBeeBitmap24 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.7.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.7.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.7.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.7.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.8 public class ZigBeeBitmap32 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.8.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.8.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.8.3 public static ZigBeeBitmap32 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.8.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.8.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.8.6 public boolean isAnalog()

- Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.8.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

- Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

  An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.9 public class ZigBeeBitmap40 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.9.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

- Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.9.2 public short getId()

- Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.9.3 public static ZigBeeBitmap40 getInstance()

- Gets a singleton instance of this class.

Returns the singleton instance.

149.34.9.4 public Class<?> getJavaDataType()

- Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.9.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.9.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.9.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.10 public class ZigBeeBitmap48 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.10.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.10.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.10.3 public static ZigBeeBitmap48 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.10.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.10.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.10.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.10.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value.*

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.11 public class ZigBeeBitmap56 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.11.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.11.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.11.3 public static ZigBeeBitmap56 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.11.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.11.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.11.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.11.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.12 public class ZigBeeBitmap64 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.12.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.12.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.12.3 public static ZigBeeBitmap64 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.12.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.12.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.12.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.12.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.13 public class ZigBeeBitmap8 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Bitmap 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.13.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.13.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.13.3 public static ZigBeeBitmap8 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.13.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.13.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.13.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.13.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.14 public class ZigBeeBoolean implements ZCLSimpleTypeDescription

A singleton class that represents the 'Boolean' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.14.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.14.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.14.3 public static ZigBeeBoolean getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.14.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.14.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.14.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.14.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.15 public class ZigBeeCharacterString implements ZCLSimpleTypeDescription

A singleton class that represents the 'Character String' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.15.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.15.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.15.3 public static ZigBeeCharacterString getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.15.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.15.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.15.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.15.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value.*

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.16 public class ZigBeeClusterID implements ZCLSimpleTypeDescription

A singleton class that represents the 'Cluster ID' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.16.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.16.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.16.3 public static ZigBeeClusterID getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.16.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.16.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.16.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.16.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.17 public class ZigBeeDate implements ZCLSimpleTypeDescription

A singleton class that represents the 'Date' data type, as it is defined in the ZigBee Cluster Library specification.

The ZigBee data type is mapped to a byte[4] array where byte[0] must contain the Year field (be careful that in the ZCL specification this byte do not contain the actual year, but an offset) whereas byte[3] the Day of Week. The array is marshaled/unmarshaled starting from byte[0].

149.34.17.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.17.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.17.3 public static ZigBeeDate getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.17.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.17.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.17.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.17.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.18 public class ZigBeeEnumeration16 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Enumeration 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.18.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.18.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.18.3 public static ZigBeeEnumeration16 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.18.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.18.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.18.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.18.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.19 public class ZigBeeEnumeration8 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Enumeration 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.19.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.19.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.19.3 public static ZigBeeEnumeration8 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.19.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.19.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.19.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.19.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.20 public class ZigBeeFloatingDouble implements ZCLSimpleTypeDescription

A singleton class that represents the 'Floating Double' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.20.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.20.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.20.3 public static ZigBeeFloatingDouble getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.20.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.20.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.20.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.20.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.21 public class ZigBeeFloatingSemi implements ZCLSimpleTypeDescription

A singleton class that represents the 'Floating Semi' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.21.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.21.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.21.3 public static ZigBeeFloatingSemi getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.21.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.21.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.21.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.21.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.22 public class ZigBeeFloatingSingle implements ZCLSimpleTypeDescription

A singleton class that represents the 'Floating Single' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.22.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.22.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.22.3 public static ZigBeeFloatingSingle getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.22.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.22.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.22.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.22.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.23 public class ZigBeeGeneralData16 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.23.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.23.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.23.3 public static ZigBeeGeneralData16 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.23.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.23.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.23.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.23.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.24 public class ZigBeeGeneralData24 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.24.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.24.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.24.3 public static ZigBeeGeneralData24 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.24.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.24.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.24.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.24.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.25 public class ZigBeeGeneralData32 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.25.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.25.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.25.3 public static ZigBeeGeneralData32 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.25.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.25.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.25.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.25.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.26 public class ZigBeeGeneralData40 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.26.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.26.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.26.3 public static ZigBeeGeneralData40 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.26.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.26.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.26.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.26.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.27 public class ZigBeeGeneralData48 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.27.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.27.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.27.3 public static ZigBeeGeneralData48 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.27.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.27.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.27.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.27.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.28 public class ZigBeeGeneralData56 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.28.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.28.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.28.3 public static ZigBeeGeneralData56 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.28.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.28.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.28.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.28.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.29 public class ZigBeeGeneralData64 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.29.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.29.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.29.3 public static ZigBeeGeneralData64 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.29.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.29.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.29.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.29.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.30 public class ZigBeeGeneralData8 implements ZCLSimpleTypeDescription

A singleton class that represents the 'General Data 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.30.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.30.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.30.3 public static ZigBeeGeneralData8 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.30.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.30.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.30.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.30.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.31 public class ZigBeeIEEE_ADDRESS implements ZCLSimpleTypeDescription

A singleton class that represents the 'IEEE ADDRESS' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.31.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.31.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.31.3 public static ZigBeeIEEE_ADDRESS getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.31.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.31.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.31.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.31.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.32 public class ZigBeeLongCharacterString implements ZCLSimpleTypeDescription

A singleton class that represents the 'Long Character String' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.32.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.32.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.32.3 public static ZigBeeLongCharacterString getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.32.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.32.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.32.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.32.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.33 public class ZigBeeLongOctetString implements ZCLSimpleTypeDescription

A singleton class that represents the 'Long Octet String' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.33.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.33.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.33.3 public static ZigBeeLongOctetString getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.33.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.33.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.33.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.33.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.34 public class ZigBeeOctetString implements ZCLSimpleTypeDescription

A singleton class that represents the 'Octet String' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.34.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.34.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.34.3 public static ZigBeeOctetString getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.34.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.34.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.34.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.34.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.35 public class ZigBeeSecurityKey128 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Security Key 128' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.35.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.35.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.35.3 public static ZigBeeSecurityKey128 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.35.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.35.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.35.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.35.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.36 public class ZigBeeSet implements ZCLDataTypeDescription

A singleton class that represents the 'Set' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.36.1 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.36.2 public static ZigBeeSet getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.36.3 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.36.4 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.36.5 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.37 public class ZigBeeSignedInteger16 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.37.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.37.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.37.3 public static ZigBeeSignedInteger16 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.37.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.37.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.37.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.37.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be `null`. If `null` a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a `null` value as the request to serialize the so called *Invalid Value*.

Throws `IOException`– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.38 public class ZigBeeSignedInteger24 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.38.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns `null` if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws `IOException`– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.38.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.38.3 public static ZigBeeSignedInteger24 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.38.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.38.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.38.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.38.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.39 public class ZigBeeSignedInteger32 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.39.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.39.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.39.3 public static ZigBeeSignedInteger32 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.39.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.39.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.39.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.39.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.40 public class ZigBeeSignedInteger40 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.40.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.40.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.40.3 public static ZigBeeSignedInteger40 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.40.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.40.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.40.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.40.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.41 public class ZigBeeSignedInteger48 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.41.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.41.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.41.3 public static ZigBeeSignedInteger48 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.41.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.41.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.41.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.41.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.42 public class ZigBeeSignedInteger56 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.42.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.42.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.42.3 public static ZigBeeSignedInteger56 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.42.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.42.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.42.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.42.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.43 public class ZigBeeSignedInteger64 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.43.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.43.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.43.3 public static ZigBeeSignedInteger64 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.43.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.43.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.43.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.43.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.44 public class ZigBeeSignedInteger8 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Signed Integer 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.44.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.44.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.44.3 public static ZigBeeSignedInteger8 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.44.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.44.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.44.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.44.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.45 public class ZigBeeStructure implements ZCLDataTypeDescription

A singleton class that represents the 'Structure' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.45.1 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.45.2 public static ZigBeeStructure getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.45.3 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.45.4 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.45.5 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.46 public class ZigBeeTimeOfDay implements ZCLSimpleTypeDescription

A singleton class that represents the 'Time Of Day' data type, as it is defined in the ZigBee Cluster Library specification. The ZigBee data type is mapped to a byte[4] array where byte[0] must contain the Hour field and byte[3] the Hundredths of seconds. The array is marshaled/unmarshaled starting from byte[0].

149.34.46.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.46.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.46.3 public static ZigBeeTimeOfDay getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.46.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.46.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.46.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.46.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.47 public class ZigBeeUnsignedInteger16 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 16-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.47.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.47.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.47.3 public static ZigBeeUnsignedInteger16 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.47.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.47.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.47.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.47.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.48 public class ZigBeeUnsignedInteger24 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 24-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.48.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.48.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.48.3 public static ZigBeeUnsignedInteger24 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.48.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.48.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.48.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.48.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.49 public class ZigBeeUnsignedInteger32 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 32-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.49.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.49.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.49.3 public static ZigBeeUnsignedInteger32 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.49.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.49.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.49.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.49.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.50 public class ZigBeeUnsignedInteger40 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 40-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.50.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.50.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.50.3 public static ZigBeeUnsignedInteger40 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.50.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.50.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.50.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.50.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.51 public class ZigBeeUnsignedInteger48 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 48-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.51.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.51.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.51.3 public static ZigBeeUnsignedInteger48 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.51.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.51.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.51.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.51.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.52 public class ZigBeeUnsignedInteger56 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 56-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.52.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.52.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.52.3 public static ZigBeeUnsignedInteger56 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.52.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.52.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.52.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.52.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.53 public class ZigBeeUnsignedInteger64 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 64-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.53.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.53.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.53.3 public static ZigBeeUnsignedInteger64 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.53.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.53.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.53.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.53.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.54 public class ZigBeeUnsignedInteger8 implements ZCLSimpleTypeDescription

A singleton class that represents the 'Unsigned Integer 8-bit' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.54.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.54.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.54.3 public static ZigBeeUnsignedInteger8 getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.54.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.54.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.54.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.54.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.34.55 public class ZigBeeUTCTime implements ZCLSimpleTypeDescription

A singleton class that represents the 'UTC Time' data type, as it is defined in the ZigBee Cluster Library specification.

149.34.55.1 public Object deserialize(ZigBeeDataInput is) throws IOException

is the ZigBeeDataInput from where the value of data type is read from.

□ Deserializes a value from the passed ZigBeeDataInput stream.

Returns An object that represents the deserialized value of data. Returns null if the read value represents the *Invalid Value* for the specific ZigBee data type.

Throws IOException– If an I/O error occurs while reading the ZigBeeDataInput. An EOFException is thrown if the data input stream end is reached while deserializing the data type.

149.34.55.2 public short getId()

□ Returns the data type identifier.

Returns the data type identifier. The data types identifiers supported by this specification are defined in the ZigBeeDataTypes interface.

149.34.55.3 public static ZigBeeUTCTime getInstance()

□ Gets a singleton instance of this class.

Returns the singleton instance.

149.34.55.4 public Class<?> getJavaDataType()

□ Returns the corresponding Java type class.

Returns the corresponding Java type class.

149.34.55.5 public String getName()

□ Returns the associated data type name.

Returns the associated data type name string.

149.34.55.6 public boolean isAnalog()

□ Checks if the data type is analog.

Returns true, if the data type is Analog, otherwise is Discrete.

149.34.55.7 public void serialize(ZigBeeDataOutput os, Object value) throws IOException

os a ZigBeeDataOutput stream where to the passed value will be appended. This parameter cannot be null. If null a NullPointerException must be thrown.

value The value that have to be serialized on the output stream. If null is passed this method outputs on the stream the ZigBee invalid value related the specific data type. If the data type do not allow any invalid value and the passed value is null an IllegalArgumentException is thrown.

□ Serializes a ZigBee data type into a ZigBeeDataOutput stream. An implementation of this method must throw an IllegalArgumentException if the passed value does not belong to the expected class or its value exceeds the possible values allowed (in terms of range or length).

An implementation of this method must interpret (where it makes sense) a null value as the request to serialize the so called *Invalid Value*.

Throws IOException– If an I/O error occurs while writing on the ZigBeeDataOutput. The EOFException may be thrown if there is no more space on the data output for serializing the passed value.

149.35 References

[1] *ZigBee Specification*
Document 053474r17, ZigBee Alliance, October 19, 2007.

[2] *ZigBee Cluster Library Specification*
Document 075123r04ZB, ZigBee Alliance, May 29, 2012.

[3] *Pervasive Service Composition in the Home Network*
André Bottaro, Anne Gérodolle, Philippe Lalanda, 21st IEEE International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 2007.

[4] *Device and Service Discovery in Home Networks with OSGi*
Pavlin Dobrev, David Famolari, Christian Kurzke, Brent A. Miller, IEEE Communications magazine, Volume 40, Issue 8, pp. 86-92, August 2002.

[5] *ASHRAE 135-2004 Standard*
Data Communication Protocol for Building Automation and Control Networks.

[6] *Listeners considered harmful: The whiteboard pattern*
Peter Kriens, BJ Hargrave for the OSGi Working Group, Technical Whitepaper, August 2004.
https://docs.osgi.org/whitepaper/whiteboard-pattern/

[7] *ZigBee Gateway*
ZigBee Alliance, 2011.

150 Configurator Specification

Version 1.0

150.1 Introduction

OSGi defines a model to provide bundles with configurations. This is specified in the Configuration Admin specification where a configuration is identified by a persistent identity (PID). A PID is a unique token, recommended to be conforming to the symbolic name syntax. A configuration consists of a set of properties, where a property consists of a string key and a corresponding value. The type of the value is limited to the primitive types and their wrappers, Strings, or Java Arrays/List/Vector of these.

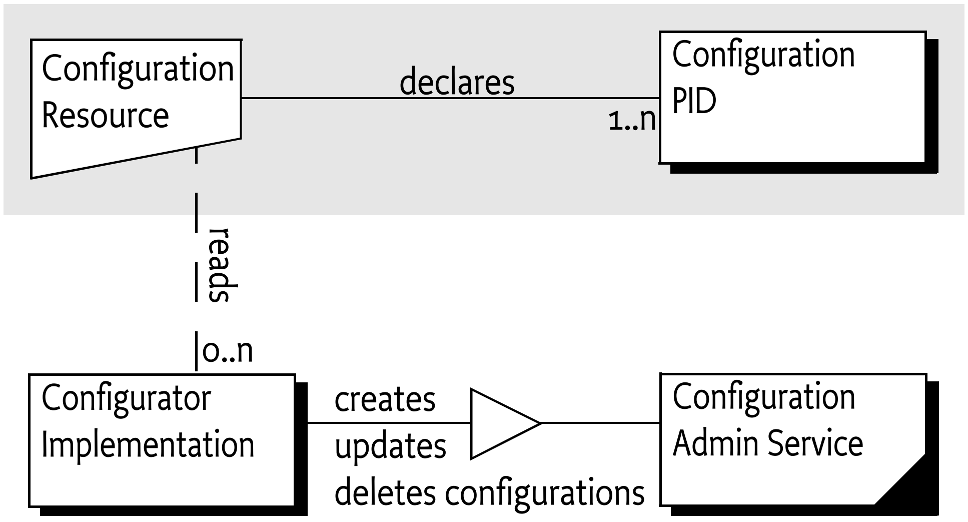
This specification defines a mechanism to feed configurations into the Configuration Admin Service through configuration resources. A single configuration resource can feed multiple PIDs with configuration and multiple configuration resources can be provided in one or more bundles.

150.2 Entities

The following entities are used in this specification.

- *Configuration Admin Service* - Standard service to configure OSGi-based systems. See *Configuration Admin Service Specification* on page 65.
- *Configuration Resource* - A JSON resource in a bundle containing configurations. This resource is processed by an implementation of this specification.
- *Extendee* - The extendee is a bundle containing configuration resources. It is *extended* by an implementation of this specification.
- *Configurator* - The Configurator implements the behavior specified in this specification. It processes configuration resources and passes the configuration dictionary on to the Configuration Admin Service.
- *Configuration dictionary* - The configuration information when it is passed to the Configuration Admin Service. It consists of a `Dictionary` object with a number of properties and identifiers.
- *Persistent Identity (PID)* - A configuration dictionary is associated with a unique PID to identify the destination of this data. See *The Persistent Identity* on page 68.
- *Configuration Object* - Implements the `Configuration` interface and contains the configuration dictionary for a Managed Service or one of the configuration dictionaries for a Managed Service Factory. These objects are manipulated by configuring bundles.
- *Coordinator Service* - The coordinator groups related operations to optimize handling of these operations. Using the coordinator with configuration updates can minimize the volatility in the system. See *Coordinator Service Specification* on page 613.

Figure 150.1 *Configurator Entity Overview*



150.3 Configuration Resources

The Configurator is processing configuration resources containing configurations. The resources can either be part of a bundle or be provided to the Configurator on startup.

150.3.1 Configuration Resource Format

The format for a configuration resource is [1] *JSON (JavaScript Object Notation)* and it must be UTF-8 encoded. An example configuration resource has the following structure:

```
{
    // Resource Format Version
    ":configurator:resource-version" : 1,

    // First Configuration
    "pid.a": {
           "key": "val",
           "some_number": 123
        },

    // Second Configuration
    "pid.b": {
           "a_boolean": true
        }
}
```

Comments in the form of [2] *JSMin (The JavaScript Minifier)* comments are supported, that is, any text on the same line after // is ignored and any text between /* */ is ignored.

Configuration resources provide a set of configuration dictionaries each with a *PID* key to target a specific PID in the Configuration Admin Service and zero or more configuration values for this PID. Keys starting with the prefix :configurator: contain information about the resource or instructions for the Configurator and therefore are not interpreted as PIDs containing configurations. If a key contains an invalid PID, this entry is ignored and the Configurator should log an error with the Log Service if available.

The Configurator defines the following special keys on the resource level.

Table 150.1 Resource-level Configurator Keys

Key	Value type	Syntax	Description
:configurator: resource-version	Number	*integer* > 0	The version of the configuration resource format. This specification only supports version 1. If this entry is omitted then version 1 is assumed. Resources specifying an unsupported or invalid version are ignored and the Configurator should log an error with the Log Service if available.
:configurator: symbolic-name	String	*symbolic-name*	The symbolic name of the configuration resource. If not specified the symbolic name of the bundle containing the resource is used. *Mandatory* for configuration resources that do not reside in a bundle.
:configurator:version	String	*version*	The version of this configuration resource. If not specified the version of the bundle containing the resource is used. *Mandatory* for configuration resources that do not reside in a bundle.

150.3.2 PIDs, Factory Configurations and Targeted PIDs

Configuration resources provide a set of configuration dictionaries each with a *PID* key to target a specific PID in the Configuration Admin Service.

Factory configurations can be addressed using the *factory PID* and a name by starting with the factory PID, appending a tilde ('~' \u007e), and then appending the name. This ensures a well-known name for the factory configuration instance. The PID for such a configuration is exactly this key. The Configurator must use the getFactoryConfiguration methods on Configuration Admin Service to create or obtain configurations with the given factory PID and name.

Targeted PIDs are supported through the configuration resource. In the case of single configurations, the full targeted PID is used as the key. For factory configurations, the key is assembled by chaining the targeted factory PID, a tilde ('~' \u007e), and the name.

The Configurator obtains all configurations with the location value of ? to allow the configurations to be received by multiple bundles.

The Configurator uses the Configuration.updateIfDifferent method on the configuration object to avoid any volatility in the system if the configuration applied has not been changed.

150.3.3 Configuration Dictionary

A configuration dictionary for the Configuration Admin Service is specified through a JSON object in the configuration resource. It is introduced using the *PID* as the key. The value is a JSON object containing the configuration dictionary.

The Configurator removes any comments and all properties where the key is starting with the special prefix :configurator: from the configuration object before converting it to a configuration dictionary that is provided to the Configuration Admin Service.

The Configurator defines special keys that can be used within the configuration object.

Table 150.2 PID-level Configurator Keys

Key	Value type	Syntax	Description
:configurator:policy	String	default or force	Specifies the overwrite policy on configurations set by non-Configurator sources. See *Overwrite Policies* on page 1150.

Key	Value type	Syntax	Description
:configurator:ranking	Number	*integer*	The ranking of this configuration. If multiple bundles provide configuration for the same PID ranking rules are used to decide which configuration gets applied, see *Ranking* on page 1149.

150.3.4 Data Types

Configuration values support data types as specified with the *Filter Syntax* in the OSGi Core Specification. Configuration resources are specified in JSON, which supports a more basic set of data types. The following table describes how values are converted from JSON to configuration values.

Table 150.3 *JSON Conversions*

JSON type	To Java type
Boolean	Boolean
Number	Whole number: Long Floating point number: Double
String	String
Array	Array, or if requested Collection. Contents are boxed. If the array contents are of the same JSON type, the associated Java type is used as the array type. Otherwise the array elements are converted to String and a String[] array is used.
Object	Literal object as JSON String

If a specific data type is required for a configuration, the Configurator can be instructed to convert the JSON value to a given data type. The target type can be specified by adding a colon : and the desired data type to the property name. Supported data types are String, Integer, Long, Float, Double, Byte, Short, Character and Boolean. Additionally arrays of Scalar or primitive types are supported and Collection of scalar. The primitive types that can be specified for arrays are: int, long, float, double, byte, short, char, boolean. For Collection the Configurator picks a suitable implementation that preserves order. Both bare Collection as well as typed collections that use the generics style notation are supported. If a requested conversion cannot be performed, then the configuration is not processed and the Configurator implementation should log an error.

An example configuration resource with typed data:

```
{
  "my.pid": {
    "port:Integer" : 300,
    "an_int_array:int[]" : [2, 3, 4],
    "an_Integer_collection:Collection<Integer>" : [2, 3, 4],
    "complex": {
      "a" : 1,
      "b" : "two"
    }
  }
}
```

The above configuration gets converted to a configuration dictionary with the following entries (in pseudo Java language):

```
Integer port = 300;
int[]   an_int_array = {2, 3, 4};
Collection<Integer> an_Integer_collection = {2, 3, 4};
```

```
String  complex = "{ \"1\" : 1, \"b\" : \"two\" }"
```

As an alternative of specifying data types for the Configurator, consumers of configuration can convert the configuration values to the desired type by using the OSGi Converter see *Converter Specification* on page 1457. A convenient way to convert a configuration map to the desired data types is by using the Converter to convert it to an annotation instance or by using a Declarative Services component property type.

150.3.4.1 Binary Data

In some cases binary data is associated with configurations such as certificates, security keys or other resources. The Configurator can manage this binary data. The bundle developer places the binaries in a location in the extendee and references it from the configuration resource, marking its type as `binary`:

```
{
  "my.config": {
    "security.enabled": true,
    "public.key:binary" : "/OSGI-INF/files/mykey.pub"
  }
}
```

When the Configurator applies the configuration, it extracts the binary file to a public area on the file system. The Configurator creates a subdirectory with as name the PID of the configuration. The PID must be URL-encoded to ensure that it does not contain characters that are illegal on a file system. The binary file is extracted in this subdirectory. The Configurator then applies the configuration with as value for the binary entry the absolute path of the extracted binary file.

A binary data property can also specify an array of binary resources by declaring the `binary[]` data type. Each resource referenced is extracted as a separate file on the file system and the value of the property will be an array of strings, each string being the full path of one extracted binary.

By default a directory called `binaries` in the bundle data area of the Configurator implementation is used. An alternative location can be specified via the `configurator.binaries` framework property. The value of this property must be an absolute path on the file system to which the Configurator has write access. If the directory does not exist the Configurator will create it. If the Configurator cannot write to this location, it logs an error and uses the default location instead.

When a configuration is removed, its associated binary files are also removed from the file system. When a configuration is updated, associated binary files are updated, if necessary. In the case of an update the Configurator should use a different filename for the extracted binary file to avoid any open file lock issues.

150.3.5 Ranking

The order in which the Configurator processes bundles is not defined. To control which configurations are in effect configuration ranking can be used. Configuration ranking is similar to service ranking; it is an integer which defaults to 0. Configurations with a higher ranking are preferred over configurations with a lower ranking. When multiple configurations arrive over time it is possible that the Configurator changes the effective configuration if a higher ranked configuration arrives later. The design of the Configurator is such that the effective set of configurations once the system stabilizes is consistent, regardless of the order in which bundles are installed and processed.

The ranking of a configuration can be specified by adding the `:configurator:ranking` property. The value of this property is converted to an `Integer` as defined by the Converter specification. If the value cannot be converted a warning should be logged. When multiple configurations for a given PID have the same ranking the bundle providing the configuration with the lowest bundle ID is preferred. If multiple configurations for the same PID with the same ranking are specified within a single bundle, the first one encountered is used.

The following example shows two bundles with a configuration resource containing a configuration for the same PID:

```
Resource in Bundle A:
{
  "my.pid": {
    "port:Integer" : 300,
    ":configurator:ranking" : 100
  }
}

Resource in Bundle B:
{
  "my.pid": {
    "port:Integer" : 100,
    ":configurator:ranking" : 10
  }
}
```

Bundle A contains the configuration with the higher ranking. Therefore, regardless of the installation order of bundle A and B, the configuration from Bundle A will be in effect after both bundles are installed and processed by the Configurator.

150.3.6 Overwrite Policies

In an IT operations scenario configurations are often updated by a systems administrator to suit the deployments requirements. In such scenarios it may be undesirable to have these modifications overwritten by a software update which includes a configuration resource. In other cases, bundles with configuration resources are used to enforce best practices or compliance with corporate guidelines, which should replace any previous manual settings. This specification defines policies to define the overwrite behavior of the Configurator when configurations have been set or modified by another entity.

Configuration policies are set by specifying the `:configurator:policy` property. Accepted values are `default` and `force`. This policy defines the behavior when a configuration to be applied was set by another entity in the system, or if it was modified by someone from the values set by the Configurator. The default value for this property is `default`. If the specified value is invalid an error is logged and the default value is used.

Table 150.4 Applying Configurations: Overwrite Policies

Policy value	Action
`default`	No action
`force`	Configuration is added

The Configurator must keep track of configuration change count values to identify configurations that were changed by other entities or administrators.

When a bundle that provides configuration resources is uninstalled, the Configurator removes any configurations that it has provided on behalf of this bundle from the system. Before it removes a configuration the Configurator checks with the Configuration Admin Service whether the configuration it has provided has been changed by another entity. If the configuration has not been changed by another entity it is removed. If it has been changed then whether the configuration is removed depends on the value of the `configurator:policy` property:

Table 150.5 Removing externally modified configurations

Policy value	Action
`default`	No action

Policy value	**Action**
force	Configuration is removed

When a configuration is removed the Configurator checks whether another, lower ranked, configuration resource is available. If present the Configurator sets this configuration.

The following examples explain the two policy options. In the first example Bundle A contains a configuration for the PID my.pid without specifying the policy. In this case the default policy is used:

```
{
  "my.pid": {
    "port:Integer" : 300
  }
}
```

The following actions demonstrate the behavior of the default policy:

1. The framework is started without any configuration for PID my.pid.
2. Bundle A is installed, the Configurator creates the configuration for PID my.pid.
3. An administrator manually changes the configuration for PID my.pid.
4. Bundle A is updated with an updated configuration for PID my.pid. The Configurator detects the manual change of the configuration in Configuration Admin Service and does not apply the updated configuration from the bundle.
5. Bundle A is uninstalled. The Configurator detects the manual change of the configuration in Configuration Admin Service and does not delete the configuration.

In the second example Bundle A contains a configuration for the PID my.pid this time with the overwrite policy set to force.

```
{
  "my.pid": {
    "port:Integer" : 300,
    ":configurator:policy" : "force"
  }
}
```

The following actions demonstrate the behavior of the force policy:

1. The framework is started without any configuration for PID my.pid.
2. Bundle A is installed, the Configurator creates the configuration for PID my.pid.
3. An administrator manually changes the configuration for PID my.pid.
4. Bundle A is updated with an updated configuration for PID my.pid. The Configurator applies the updated configuration.
5. Bundle A is uninstalled. The Configurator detects the manual change of the configuration in Configuration Admin Service and deletes the configuration.

150.4 Bundle Configuration Resources

The Configurator follows the OSGi extender model and looks for JSON configuration resources in installed bundles, if the bundle has opted-in to be processed. In order to get processed, a bundle must require the Configurator extender:

```
Require-Capability: osgi.extender;
  filter := "(&(osgi.extender=osgi.configurator)
        (version>=1.0)(!(version>=2.0)))"
```

The Configurator must ensure to only process bundles that it is wired to by the resolver.

By default the configuration resources are in the `OSGI-INF/configurator` directory in the bundle.

Configuration files are UTF-8 encoded and have the `.json` file extension. Files not having this extension are ignored. The Configurator processes the configuration resources within a single bundle in lexical order using the full resource path for sorting.

150.5 Initial Configurations

When the Configurator starts it calls `bundleContext.getProperty("configurator.initial")` to obtain initial configurations from the runtime environment. If this property is available its value is processed as follows:

1. If the value starts with a left curly bracket ('{' `\u007B`), ignoring any leading white space, the Configurator will interpret the value as a literal configuration JSON resource.
2. Otherwise the value is treated as a comma-separated list of URLs. The Configurator will read the resource at each URL and parse it as a JSON Configuration resource. If any errors occur during this process they are logged and the URL is skipped. The URLs are processed in alphabetical order of their provided value.

The ranking of these configurations can be set in the configuration resource as described in *Ranking* on page 1149. The Configurator treats the initial configurations as being provided from a bundle with the bundle id `-1`.

If the framework is restarted, the Configurator needs to check whether the provided initial configurations are different than on the previous startup. The implementation is free to use whatever is appropriate to perform this check, like comparing last modified for the URLs or using a hash etc. If the provided configuration is different than on a previous startup, this is treated like a bundle update with an updated configuration.

150.6 Life Cycle

The Configurator uses the Configuration Admin Service. Therefore the Configurator implementation should require the Configuration Admin Service through a service requirement. The Configurator should not start processing configuration resources until it has runtime access to the Configuration Admin Service.

The Configurator uses the Configuration Admin Service that is visible to both the Configurator itself as well as the bundle that is being processed. If there are multiple candidates, the service with the highest ranking is used. If there is no Configuration Admin Service visible to both the bundle that is processed and the Configurator, the processing is delayed until such a service becomes available.

When the Configurator starts, it processes all started bundles and applies configurations provided by those bundles. From then on, the Configurator processes bundles as they enter the `STARTING` state. The Configurator should process as many bundles as possible in a single pass to minimize volatility for PIDs where multiple configurations with different rankings are provided.

When a bundle containing configuration resources is updated, the configurations must be updated in the Configuration Admin Service to which they were originally provided, keeping in mind that the system might have been restarted in-between. One way of keeping track of the original Configuration Admin Service is via the bundle location of the bundle providing the service. If this service is not available the Configurator must attempt to apply the updated configuration when this Configuration Admin Service re-appears.

Configurations remain in the system until the bundle that provided the configurations is uninstalled. When this happens, the Configurator must uninstall the configurations from the Configuration Admin Service to which it originally installed it as is the case with updates. If this Configuration Admin Service is not available at this time, the Configurator must remember the configurations that are to be removed, and remove them when the Configuration Admin Service re-appears at a later time.

When the Configurator becomes active, it must check whether configurations that it installed previously are still valid. If the bundles that provided these configurations have been uninstalled, the associated configurations must be removed. If a bundle is updated the associated configurations are also updated. The Configurator calls `updateIfDifferent` on the configuration to avoid volatility in the system if the actual configuration values did not change.

When updating or removing configurations, the Configurator must take the *Overwrite Policies* on page 1150 into account. This means that for certain policy values an externally modified configuration is not replaced or removed.

When a bundle that provides the Configuration Admin Service is uninstalled, the Configurator considers all configurations previously provided to that Configuration Admin Service as not yet applied. If another Configuration Admin Service is or becomes visible to both the Configurator and the bundle containing configuration resources, the Configurator will provide the configurations to this Configuration Admin Service as new.

When the Configurator is stopped or uninstalled the configurations applied will remain in the system.

150.7 Grouping and Coordinations

The *Coordinator Service Specification* on page 613 defines a mechanism for multiple parties to collaborate on a common task without *a priori* knowledge of who will collaborate in that task. The Configurator must participate in such scenarios to coordinate with provisioning or configuration tasks.

Whenever the Configurator is processing configuration resources and interacting with the Configuration Admin Service, the Configurator must check whether a Coordinator Service is present. If it is present, the Configurator checks for an implicit coordination on the current thread. If such an implicit coordination exists, the Configurator does not need to create one. However, if such an implicit coordination is not present, the Configurator starts an implicit coordination on the current thread when interacting with the Configuration Admin Service and ends this coordinator when it is finished doing the current set of work. The Configurator does not need to delay applying any changes to the Configuration Admin Service until the coordination ends.

150.8 Security

When Java permissions are enabled, the Configurator must perform the following security procedures.

150.8.1 Configuration Permission

The Configurator manages configurations on behalf of the bundle containing the configuration resources. Therefore the Configurator needs to have the `ConfigurationPermission[*,org.osgi.service.cm.ConfigurationPermission.CONFIGURE]`.

Every bundle has the implicit right to receive and configure configurations with a location that exactly matches the Bundle's location or that is null. Therefore the extendee does not need to special permissions.

150.8.2 Service Permission

The Configurator needs ServicePermission[<interface>, GET] for the Coordinator service.

The extendee needs ServicePermission[<interface>, GET] for the Configuration Admin Service.

150.8.3 Configuration Admin Service

The Configurator does get the Configuration Admin Service on behalf of the extendee. Therefore the extendee needs to be included in permission checks for getting the Configuration Admin Service. The Configurator needs to perform the required calls to ensure the extendee has the necessary permission to get the Configuration Admin Service.

150.8.4 File Permission

If binaries are used, the Configurator needs to have read/write/delete permission to the configured directory to store the binaries.

A bundle using a binary referenced from a configuration needs to have read permission to correct sub directory of the configured binary directory. The subdirectory is named after the PID of the configuration.

By default binaries are stored in the bundle data are of the Configurator. While this works without Java security enabled, permission configuration for the extendees gets challenging as the location of the bundle data area is only known at runtime. Therefore with Java security enabled, the directory holding the binaries should be configured to allow permission configuration for the extendees.

150.9 Capabilities

150.9.1 osgi.extender Capability

The Configurator implementation bundle must provide the osgi.extender capability with name osgi.configurator with the version of this specification:

```
Provide-Capability: osgi.extender;
     osgi.extender="osgi.configurator";
     version:Version="1.0"
```

This capability must follow the rules defined for the *osgi.extender Namespace* on page 707.

Bundles providing configuration resources must require the osgi.extender capability to opt in to being processed by the Configurator. The default location for configuration resources is in OSGI-INF/configurator. A bundle can specify alternate locations for configuration resources through the configurations attribute. The value of this attribute is of type String or List<String>. Each value represents a path inside the bundle. This path is always relative to the root of the bundle and may start with a slash /. A path value of / indicates the root of the bundle. The Configurator uses Bundle.findEntries to find all resources with the .json extension in this location. Sub directories are not considered. If the configuration attribute specifies multiple paths, these are visited in the order specified. Duplicate paths are ignored. Paths that do not exist in the bundle are logged as an error and skipped. Resources in a single directory are processed in alphabetical order. For example:

```
Require-Capability: osgi.extender;
     filter:="(&(osgi.extender=osgi.configurator)
             (version>=1.0)(!(version>=2.0)))";
```

```
configurations="resources/configs"
```

To simplify the creation of this requirement the RequireConfigurator annotation can be used. This annotation allows the configurations attribute to be defined is a value other than the default is needed.

```
@RequireConfigurator("resources/configs")
```

150.10 osgi.configuration Namespace

Configuration resources define configuration for one or more PIDs. To declare what configuration is being provided, the osgi.configuration capability namespace can be used. Configuration resources and bundles can define the osgi.configuration capability for each configuration that they define. This capability should have resolve time effectiveness.

The osgi.configuration Namespace supports the attributes defined in the following table and ConfigurationNamespace.

Table 150.6 osgi.configuration namespace definition

Name	Kind	M/O	Type	Syntax	Description
service.pid	CA	O†	String	qname	Defines the PID of the configuration.
service.factoryPid	CA	O†	String	qname	Defines the factory PID if this is a factory configuration.

† Note that at least one of service.pid or service.factorypid must be defined. If the configuration is a standard configuration then only the service.pid is used. If the configuration is a factory configuration with an automatically generated identity then only the service.factoryPid is used. If the configuration is a factory configuration with a specified identity then both the service.pid and service.factoryPid are used.

150.11 Configuration Resources in a Repository

The configuration file format in *Configuration Resources* on page 1146 defines a portable representation of configurations for the Configuration Admin Service. Whilst the Configurator implementation is necessary to process these configurations when they are packaged inside a bundle or provided on startup, these files can also offer significant value to other tools for deployment and management outside of the Configurator usage.

If configuration resources are used in an OSGi repository, in order to integrate with querying and the resolution process, the configuration resources should define the appropriate capabilities.

In addition to the common requirements and capabilities, a standalone configuration resource must declare the following capabilities when in an OSGi repository:

- An osgi.content capability. The mime type of the configuration resource should be application/vnd.osgi.configuration+json.
- An osgi.identity capability. This capability requires that each resource define a symbolic name and version. These can be obtained from the mandatory :configurator:symbolic-name and :configurator:version keys in the configuration resource. As type attribute the string osgi.configuration must be used.

150.12 org.osgi.service.configurator

Configurator Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.configurator; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.configurator; version="[1.0,1.1)"`

150.12.1 Summary

- `ConfiguratorConstants` - Defines standard constants for the Configurator services.

150.12.2 public final class ConfiguratorConstants

Defines standard constants for the Configurator services.

150.12.2.1 public static final String CONFIGURATOR_BINARIES = "configurator.binaries"

Framework property specifying the directory to be used by the Configurator to store binary files.

If a value is specified, the Configurator will write all binaries to the given directory. Therefore the Configurator bundle needs read and write access to this directory.

If this property is not specified, the Configurator will store all binary files in its bundle private data area.

150.12.2.2 public static final String CONFIGURATOR_EXTENDER_NAME = "osgi.configurator"

The name of the extender capability attribute for the Configurator

150.12.2.3 public static final String CONFIGURATOR_INITIAL = "configurator.initial"

Framework property specifying initial configurations to be applied by the Configurator on startup.

If the value of this property starts with a '{' (ignoring leading whitespace) it is interpreted as JSON and directly feed into the Configurator.

Otherwise the value is interpreted as a comma separated list of URLs pointing to JSON documents.

150.12.2.4 public static final String CONFIGURATOR_SPECIFICATION_VERSION = "1.0"

The version of the extender capability for the Configurator specification

150.12.2.5 public static final String POLICY_DEFAULT = "default"

Value for defining the default policy.

See Also PROPERTY_POLICY

150.12.2.6 public static final String POLICY_FORCE = "force"

Value for defining the force policy.

See Also PROPERTY_POLICY

150.12.2.7 public static final String PROPERTY_POLICY = ":configurator:policy"

Configuration property for the configuration policy.

Allowed values are POLICY_DEFAULT and POLICY_FORCE

See Also POLICY_DEFAULT, POLICY_FORCE

150.12.2.8 public static final String PROPERTY_PREFIX = ":configurator:"

Prefix to mark properties as input for the Configurator when processing a configuration resource.

150.12.2.9 public static final String PROPERTY_RANKING = ":configurator:ranking"

Configuration property for the configuration ranking.

The value of this property must be convertible to a number.

150.12.2.10 public static final String PROPERTY_RESOURCE_VERSION = ":configurator:resource-version"

Global property in the configuration resource specifying the version of the resource format.

Currently only version 1 is defined for the JSON format and therefore the only allowed value is 1 for this property. If this property is not specified, 1 is assumed.

150.12.2.11 public static final String PROPERTY_SYMBOLIC_NAME = ":configurator:symbolic-name"

Global property in the configuration resource specifying the symbolic name of the configuration resource. If not specified the symbolic name of the bundle containing the resource is used. Mandatory for configuration resources that do not reside in a bundle

150.12.2.12 public static final String PROPERTY_VERSION = ":configurator:version"

Global property in the configuration resource specifying the version of the resource. If not specified the version of the bundle containing the resource is used. Mandatory for configuration resources that do not reside in a bundle.

150.13 org.osgi.service.configurator.annotations

Configurator Annotations Package Version 1.0.

This package contains annotations that can be used to require the Configurator extender.

Bundles should not normally need to import this package as the annotations are only used at build-time.

150.13.1 Summary

- `RequireConfigurator` - This annotation can be used to require the Configurator extender.

150.13.2 @RequireConfigurator

This annotation can be used to require the Configurator extender. It can be used directly, or as a meta-annotation.

This annotation allows users to define custom locations that should be searched for configuration files using RequireConfigurator.value()

Retention `CLASS`

Target `TYPE`, `PACKAGE`

150.13.2.1 String[] value default {}

□ This attribute can be used to define one or more locations that the configurator must search, in order, for configuration files.

If no locations are defined then the Configurator default of `/OSGI-INF/configurator` will be used.

Returns A list of bundle locations containing configuration files

150.14 org.osgi.service.configurator.namespace

Configurator Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Working Group.

150.14.1 Summary

- `ConfigurationNamespace` - Configuration Capability and Requirement Namespace.

150.14.2 public final class ConfigurationNamespace extends Namespace

Configuration Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type `String` and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type `String`, unless otherwise indicated.

Concurrency Immutable

150.14.2.1 public static final String CONFIGURATION_NAMESPACE = "osgi.configuration"

Namespace name for configuration capabilities and requirements.

Also, the capability attribute used to specify the name of the extension.

150.14.2.2 public static final String FACTORY_PID_ATTRIBUTE = "service.factoryPid"

The capability attribute contains the factory PID if this is a factory configuration. The value of this attribute must be of type String.

150.14.2.3 public static final String SERVICE_PID_ATTRIBUTE = "service.pid"

The capability attribute contains the PID of the configuration. The value of this attribute must be of type String.

150.15 References

[1] *JSON (JavaScript Object Notation)*
https://www.json.org

[2] *JSMin (The JavaScript Minifier)*
https://www.crockford.com/javascript/jsmin.html

151 Whiteboard Specification for Jakarta™ RESTful Web Services

Version 2.0

151.1 Introduction

REpresentational State Transfer (REST) is a simple pattern for producing Web Services. RESTful services use URI pattern matching to match a particular web resource. Different HTTP verbs, for example GET and DELETE, map to different operations on that resource. Standard HTTP response codes are used to communicate the result of an operation, potentially including a response body if the operation returns a result.

The [1] *Jakarta RESTful Web Services 3.0 Specification* defines a set of annotation mappings which allow Plain Old Java Objects (POJOs) to be directly exposed as RESTful web resources; these resources can also be grouped together using a Jakarta RESTful Web Services Application. Furthermore the specification defines a plugable model for extending the behavior of the application and the features of the Jakarta RESTful Web Services container itself. For example an extension may define specific error responses that should be sent when particular exceptions occur, or an extension may add support for serializing responses to a different format. The OSGi Whiteboard Specification for Jakarta RESTful Web Services provides a light and convenient way of using these POJOs, applications and extensions in an OSGi environment through the use of the [3] *Whiteboard Pattern*.

The Whiteboard Specification for Jakarta RESTful Web Services supports:

- *Registering Resources* - Registering a Jakarta RESTful Web Services annotated POJO in the Service Registry makes it available to be bound to an endpoint and to start responding to incoming requests.
- *Registering Applications* - Registering a Jakarta RESTful Web Services Application in the Service Registry makes it available to be bound to an endpoint and to start responding to incoming requests.
- *Registering Extensions* - The Jakarta RESTful Web Services specification defines a variety of plugable extensions. These extensions can be registered in the Service Registry to include them in the handling pipeline.
- *Requiring Extensions* - Sometimes Jakarta RESTful Web Services resources, or even Jakarta RESTful Web Services extensions, depend upon the presence of another extension. For example a Jakarta RESTful Web Services resource and exception mapper may both depend on a JSON serializer. Jakarta RESTful Web Services Whiteboard services may define preconditions that must be satisfied before they can be bound.

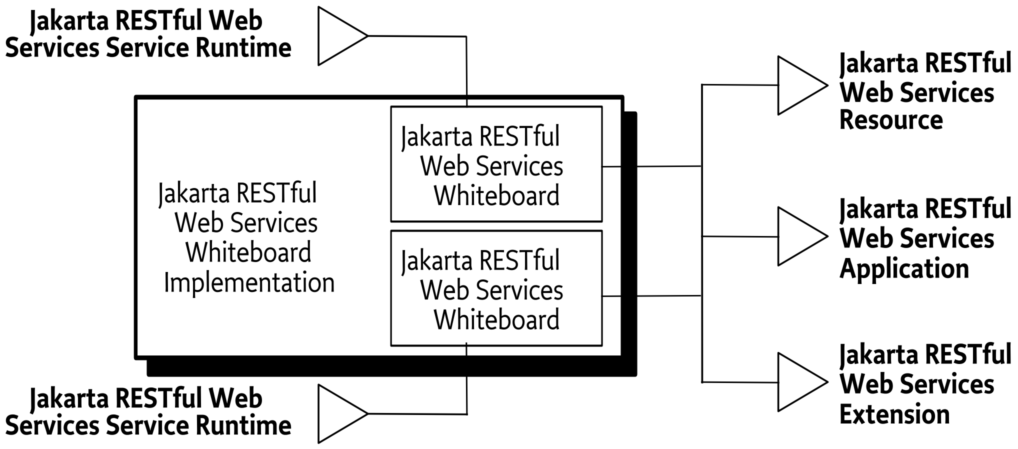
Jakarta RESTful Web Services Whiteboard implementations must support at least version 3.0 of the Jakarta RESTful Web Services API.

151.1.1 Entities

This specification defines the following entities:

- *Jakarta RESTful Web Services Whiteboard service* - An object registered in the Service Registry providing the necessary Whiteboard service properties defined by this specification. Whiteboard services may be *resource*, *application* or *extension* services
- *Jakarta RESTful Web Services Whiteboard implementation* - An implementation that provides one or more Jakarta RESTful Web Services Whiteboards.
- *Jakarta RESTful Web Services Whiteboard* - A runtime instance that processes Jakarta RESTful Web Services Whiteboard services. Each Jakarta RESTful Web Services Whiteboard service may be processed by multiple Jakarta RESTful Web Services Whiteboards. Different Jakarta RESTful Web Services Whiteboards provided by the same Jakarta RESTful Web Services Whiteboard implementation may configured differently, for example using different ports or root contexts.
- *Jakarta RESTful Web Services Service Runtime service* - A service providing runtime introspection into a Jakarta RESTful Web Services Whiteboard instance.
- *Jakarta RESTful Web Services Resource Service* - A service that provides one or more RESTful resource methods which map to incoming HTTP requests.
- *Jakarta RESTful Web Services Application Service* - A service that provides a `jakarta.ws.rs.core.Application` to be hosted by a Jakarta RESTful Web Services Whiteboard.
- *Jakarta RESTful Web Services Extension Service* - A service that extends the functionality of a Jakarta RESTful Web Services Whiteboard.
- *Static Resources* - Jakarta RESTful Web Services resources that are included programmatically in a Jakarta RESTful Web Services Whiteboard application, rather than being added at runtime by the whiteboard.

Figure 151.1 Jakarta RESTful Web Services Whiteboard Overview Diagram


Jakarta RESTful Web Services Service Runtime
Jakarta RESTful Web Services Whiteboard Implementation
Jakarta RESTful Web Services Whiteboard
Jakarta RESTful Web Services Whiteboard
Jakarta RESTful Web Services Service Runtime
Jakarta RESTful Web Services Resource
Jakarta RESTful Web Services Application
Jakarta RESTful Web Services Extension

The Figure 151.1 shows an OSGi framework running a Jakarta RESTful Web Services Whiteboard Implementation bundle. This bundle has been configured to provide two Jakarta RESTful Web Services whiteboards, each of which has a corresponding Jakarta RESTful Web Services Service Runtime Service. The various Jakarta RESTful Web Services Whiteboard services available in the framework are discovered and processed by both whiteboards.

151.2 The Jakarta RESTful Web Services Whiteboard

An important principle of the Whiteboard Specification for Jakarta RESTful Web Services is that an OSGi framework may contain many active Jakarta RESTful Web Services Whiteboards at any time, even if there is only a single Jakarta RESTful Web Services Whiteboard implementation present in the framework. In addition to providing a web endpoint with which to register Whiteboard services, a Jakarta RESTful Web Services Whiteboard provides a holder for Jakarta RESTful Web Services Applications.

All Jakarta RESTful Web Services Whiteboards have a default application which is used to register resources that do not target an existing application. In this respect a Jakarta RESTful Web Services whiteboard application shares some similarities with a Servlet Context in the *Jakarta Servlet Whiteboard* on page 785. Resources registered with a Jakarta RESTful Web Services Whiteboard are always registered as part of an application. The generated name of the default application is .default, and it is mapped to the root context of the Jakarta RESTful Web Services Whiteboard.

A Jakarta RESTful Web Services Whiteboard implementation must create a Jakarta RESTful Web Services Whiteboard instance, however it is expected that most implementations will permit multiple Jakarta RESTful Web Services whiteboards to be configured. These instances may differ significantly, or may simply offer the same capabilities on a different port.

For details on the association process between Jakarta RESTful Web Services Whiteboard services and a Jakarta RESTful Web Services Whiteboard see *Common Whiteboard Properties* on page 791.

151.2.1 The Jakarta RESTful Web Services Service Runtime Service

The JakartarsServiceRuntime service represents the runtime state information of a Jakarta RESTful Web Services Whiteboard instance. This information is provided through Data Transfer Objects (DTOs). The architecture of OSGi DTOs is described in *OSGi Core Release 8*.

Each Jakarta RESTful Web Services Whiteboard implementation registers exactly one JakartarsServiceRuntime service per Jakarta RESTful Web Services Whiteboard. The service properties of the Jakarta RESTful Web Services Service Runtime Service can be used to target Jakarta RESTful Web Services Whiteboard services at specific Jakarta RESTful Web Services whiteboards, as described by the osgi.jakartars.whiteboard.target property in *Common Whiteboard Properties* on page 1163.

The JakartarsServiceRuntime provides service registration properties to declare its underlying Jakarta RESTful Web Services Whiteboard. These service properties can include implementation-specific key-value pairs. They also include the following:

Table 151.1 Service properties for the JakartarsServiceRuntime service

Service Property Name	**Type**	**Description**
osgi.jakartars.endpoint	String+	Endpoint(s) where this Jakarta RESTful Web Services Whiteboard is listening. Registered Whiteboard services are made available here. Values could be provided as URLs e.g. http://192.168.1.10:8080/ or relative paths, e.g. /myapp/. Relative paths may be used if the scheme and authority parts of the URLs are not known, for example if the Jakarta RESTful Web Services Whiteboard is delegating to a bridged Http Service implementation. If the Jakarta RESTful Web Services Whiteboard Service is serving the root context and scheme and authority are not known, the value of the property is /. Each entry must end with a slash. See JAKARTA_RS_SERVICE_ENDPOINT.
service.changecount	Long	Whenever the DTOs available from the Jakarta RESTful Web Services Service Runtime service change, the value of this property will increase. This allows interested parties to be notified of changes to the DTOs by observing Service Events of type MODIFIED for the JakartarsServiceRuntime service. See org.osgi.framework.Constants.SERVICE_CHANGECOUNT in *OSGi Core Release 8*.

151.2.2 Inspecting the Runtime DTOs

The Jakarta RESTful Web Services Service Runtime service provides information about registered Whiteboard services through the RuntimeDTO.

The Runtime DTO provides information about services that have been successfully registered as well as information about the Jakarta RESTful Web Services Whiteboard services that were not successfully registered. Jakarta RESTful Web Services Whiteboard services that have the required properties set but cannot be processed, are reflected in the failure DTOs. Jakarta RESTful Web Services Whiteboard services of interfaces described in this specification that do not have the required properties set are ignored and not reflected in the failure DTOs.

The Runtime DTO can be obtained using the getRuntimeDTO() method. The Runtime DTO returned provides a snapshot of the state of the Jakarta RESTful Web Services Runtime, including the Jakarta RESTful Web Services Whiteboard resources, extensions and applications that are active in each registered application. The Runtime DTO also includes information about Whiteboard services which could not be activated.

151.2.2.1 DTO properties

When whiteboard services are registered with the whiteboard they must be introspected and this information reflected in the DTO(s) for that service. This introspection will include looking for annotations such as @GET and @Path both at a class and method level. The values associated with these annotations must then be appropriately combined, for example when @Path is declared on a type and method level, and recorded in the DTO.

151.2.2.2 Failure DTOs

There are a variety of reasons that whiteboard services may not be able to be used by the whiteboard. For example, if the whiteboard service cannot be retrieved from the service registry, or if the whiteboard service provides an invalid service property value, such as a malformed filter.

In these cases the failed services are represented in the Runtime DTO under one of the failed DTO properties. Depending upon the failure reason one or more of the properties of the failed DTO may be unavailable. For example if the service cannot be retrieved from the service registry then it cannot be introspected for annotations. A failure DTO will always contain the service id for the failed service and the failure reason. The whiteboard implementation must then fill in other DTO properties on a best effort basis.

151.2.3 Relation to the Servlet Container

Implementations of this specification will often be backed by existing servlet containers, such as the OSGi Http Whiteboard, or a Jakarta EE application server. There may also exist implementations which bridge into a servlet container into which the OSGi Framework has been deployed as a Web Application.

In bridged situations the Jakarta RESTful Web Services Whiteboard implementation will have limited facilities for creating new Jakarta RESTful Web Services whiteboards, and may also have limited information about its environment.

Information about the surrounding Servlet Container, including ServletContext information and HttpSession data, is available to Jakarta RESTful Web Services Whiteboard resources using standard Jakarta RESTful Web Services injection behavior.

```
@GET
@Path("{name}")
public String interrogateSession(@PathParam("name") String name,
         @Context HttpServletRequest req) {
    HttpSession s = req.getSession();
    return String.valueOf(s.getAttribute(name));
}
```

A Jakarta RESTful Web Services Whiteboard implementation needs to ensure that Http Sessions are not shared amongst different Jakarta RESTful Web Services Whiteboards, or amongst different Jakarta RESTful Web Services Whiteboard applications. That is, `HttpServletRequest.getSession()` calls must provide different sessions for each whiteboard application with which a Jakarta RESTful Web Services whiteboard service is associated.

151.2.4 Isolation between Jakarta RESTful Web Services Whiteboards

Even when they are created by the same Jakarta RESTful Web Services Whiteboard implementation, each Jakarta RESTful Web Services Whiteboard instance is separate, and isolated from other instances. Importantly, Jakarta RESTful Web Services Whiteboard services targeted to one Jakarta RESTful Web Services Whiteboard application must not be visible in any other Whiteboard or applications to which they are not targeted.

This isolation restriction is critical, as it ensures that different Jakarta RESTful Web Services Whiteboard applications can be configured with different, potentially overlapping, incompatible extension features.

151.3 Common Whiteboard Properties

Jakarta RESTful Web Services Whiteboard services support common service registration properties to associate them with a Jakarta RESTful Web Services Whiteboard. These properties apply to whiteboard resources, extensions and applications except where explicitly stated otherwise. Each service property has an associated Component Property Type annotation that can be used to easily apply the property to a Declarative Services Component.

Table 151.2 Common properties

Service Property	Type	Description

`osgi.jakartars.name`
 	
`JakartarsName`
	`String`
	
	optional
		A user defined name that can be used to identify a Jakarta RESTful Web Services whiteboard service. Names must follow OSGi symbolic name rules, and also must not start with the prefixes '.' or 'osgi.'.
		
		If no name is defined for a Jakarta RESTful Web Services whiteboard service then one is generated for it. This generated name will start with a '.'. The prefix `osgi.` is currently unused, but reserved for future versions of this specification.
		
		If a Jakarta RESTful Web Services service is registered with an illegal name then it is not bound and this is reflected in the failure DTOs. If two Jakarta RESTful Web Services services are registered with the same name (even if they are advertised as different types) then only the first service in ranking order, as specified in `ServiceReference.compareTo`, is bound and the lower ranked service(s) are reflected in the failure DTOs. See `JAKARTA_RS_NAME`.

Service Property	Type	Description

osgi.jakartars.application.select†
 JakartarsApplicationSelect
	String+
	 optional
		One or more LDAP-style filters to select the Jakarta RESTful Web Services Application(s) with which this Whiteboard service should be associated. Any service property of the Application can be filtered on. If these filters are not defined then the default Application is used. If multiple filters are defined then each filter is used to select target applications. Target applications are selected at most once, even if they match more than one filter. The default application can also be specifically targeted using the application name .default.
		
		 For example, to select an Application with name myApp provide the following filter:
		
		 `(osgi.jakartars.name=myApp)`
		
		 To select all Applications in the whiteboard provide the following value:
		
		 `(osgi.jakartars.name=*)`
		
		 If no matching application exists this is reflected in the failure DTOs. See JAKARTA_RS_APPLICATION_SELECT.
		
		 † Note that this property is not valid for Jakarta RESTful Web Services Application services.

osgi.jakartars.extension.select JakartarsExtensionSelect	String+ *optional*	A set of LDAP-style filters used to express dependencies on one or more extension services. If a filter is provided then the Jakarta RESTful Web Services Whiteboard attempts to match that filter against the service properties of the Whiteboard runtime, the service properties of the whiteboard application, and each of the extension services currently active in the application. This search may occur in any order. If all of the supplied filters are matched then the whiteboard service is registered into the Jakarta RESTful Web Services Whiteboard application. For example, to require an extension which provides JSON serialization advertising property name serialize.to with value JSON provide the following filter: `(serialize.to=JSON)` A more detailed version of this example is available in *A Jakarta RESTful Web Services Whiteboard Extension Example* on page 1173 If any filter(s) fail to match then this is reflected in the failure DTOs. See JAKARTA_RS_EXTENSION_SELECT.
osgi.jakartars.whiteboard.target JakartarsWhiteboardTarget	String *optional*	The value of this service property is an LDAP-style filter expression to select the Jakarta RESTful Web Services Whiteboard(s) to handle this Whiteboard service. The LDAP filter is used to match JakartarsServiceRuntime services. Each Jakarta RESTful Web Services Whiteboard exposes exactly one JakartarsServiceRuntime service. This property is used to associate the Whiteboard service with the Jakarta RESTful Web Services Whiteboard that registered the JakartarsServiceRuntime service. If this property is not specified then the service will target all Jakarta RESTful Web Services Whiteboards. See JAKARTA_RS_WHITEBOARD_TARGET.

151.4 Registering RESTful Resources

Jakarta RESTful Web Services resources can be registered with the Jakarta RESTful Web Services Whiteboard by registering them as Whiteboard services. This means that the resource POJO implementations are registered in the Service Registry. As Jakarta RESTful Web Services resources are POJOs they may be registered using *any* valid service interface, including Object. The Jakarta RESTful Web Services container will then use reflection to discover methods and annotations on the resource object, just as it would outside of OSGi.

As Jakarta RESTful Web Services resources have no common interface type they are instead registered with the osgi.jakartars.resource service property with a value of "true". This property serves as a marker to the Jakarta RESTful Web Services whiteboard runtime, indicating that this OSGi service should be hosted as a Jakarta RESTful Web Services Whiteboard resource.

151.4.1 Jakarta RESTful Web Services Resource mapping

Jakarta RESTful Web Services resources use the Path annotation to bind themselves to particular URIs within the Jakarta RESTful Web Services container. The path annotation can be applied to the resource class, and to individual resource methods. For example the following Jakarta RESTful Web Services resource:

```
@Path("foo")
public class Foo {

    private final List<String> entries =
        Arrays.asList("fizz", "buzz", "fizzbuzz");

    @GET
    public List<String> getFoos() {
        return Collections.unmodifiableList(entries);
    }

    @GET
    @Path("{name}")
    public String getFoo(@PathParam("name") String name) {
        if(entries.contains(name)) {
            return "A foo called " + name;
        }
        throw new IllegalArgumentException(“No foo called “ + name);
    }

}
```

This Jakarta RESTful Web Services resource defines two resource methods. The Path annotation applied to the class sets the base URI for all methods in the resource. The getFoos() method is therefore bound to the URI foo. The Path annotation on the getFoo() method makes this method a subresource which captures the next token in the URI. This method is therefore bound to URIs of the form foo/buzz.

When used as an OSGi Jakarta RESTful Web Services Whiteboard service a Jakarta RESTful Web Services resource follows the same mapping rules, but the base context(s) it uses are determined by the Application(s) to which it is mapped. For example, when mapped to the default application of a whiteboard with endpoint http://127.0.0.1/ the getFoos() method would be available at http://127.0.0.1/foo.

151.4.1.1 Clashing resource mappings

Resource services bound to a Jakarta RESTful Web Services whiteboard application share a single URI namespace with other resources in the application (including any existing static resources). When Jakarta RESTful Web Services services are bound it is possible that one or more methods on these services will map to the same URI. This situation is permitted by the Jakarta RESTful Web Services specification which defines a detailed selection algorithm.

When clashes occur in the Jakarta RESTful Web Services whiteboard then resources supplied using the service whiteboard must be preferred to static resources contained in the application. If two or more whiteboard resources exist then they must be ordered using ranking order, as specified in `ServiceReference.compareTo`. Unlike for other services in the Jakarta RESTful Web Services whiteboard, whiteboard resource services must not be ordered using their natural ordering. Whiteboard resource services with the same ranking must be considered equal, following the normal resource method selection rules defined in the Jakarta RESTful Web Services specification. As per the Core specification, whiteboard services with no `service.ranking` property must be treated as having a ranking of 0.

151.4.2 Jakarta RESTful Web Services Whiteboard Resource Lifecycle

A key tenet of Jakarta RESTful Web Services is that all resource objects are stateless. In the Jakarta RESTful Web Services specification resources therefore have one of two scopes, they are either singleton, or request-scoped. Singleton resources are created once, potentially outside the Jakarta RESTful Web Services container, and request-scoped resources are created on-demand for each request, then discarded afterwards.

Typically Jakarta RESTful Web Services developers are encouraged to write request-scoped resources, as this makes it difficult to accidentally write stateful components. In OSGi, however, it is more common to write singleton services. On demand instances of OSGi services can be created, but only if the service is registered as a `prototype` scope.

The Jakarta RESTful Web Services whiteboard implementation is responsible for managing the mismatch between the OSGi service lifecycle model and the Jakarta RESTful Web Services resource lifecycle model. If the Jakarta RESTful Web Services whiteboard resource is registered as prototype scope then the implementation must treat the resources as request-scoped, creating a new service instance for each request and releasing it when the request completes. Otherwise the Jakarta RESTful Web Services whiteboard service must be registered as a singleton scope resource within the application. Singleton scope whiteboard resources must be released by the Jakarta RESTful Web Services whiteboard when the application with which they have been registered is removed from the whiteboard, even if this is only a temporary situation.

If a failure occurs when getting the resource service this will prevent the service from being used, which is reflected using a failure DTO. In such a case the system treats the resource as unusable.

When multiple Jakarta RESTful Web Services Whiteboard implementations are present all of them can potentially process the whiteboard resources. In such situations it can be useful to associate the servlet with a specific whiteboard by specifying the `osgi.http.whiteboard.target` property on the service.

151.4.2.1 Resource Context Injection

Jakarta RESTful Web Services resources may have objects injected into them by the Jakarta RESTful Web Services container. These objects may be related to an incoming request, for example an HTTP header value, or part of the container runtime. Injected resources are annotated with a Jakarta RESTful Web Services annotation, for example `@Context`, and may be injected as method parameters, or as fields in the object.

If the Jakarta RESTful Web Services injected objects are passed as method parameters then the resource object may be a singleton. If, however, the objects are injected into fields by the Jakarta RESTful Web Services container then the resource should be declared as a `prototype` scope. Jakarta RESTful Web Services Whiteboard implementations may support field injection for singleton resources, however this behavior is non portable, and may lead to errors at runtime when using other implementations.

151.4.2.2 Request-Scoped Resources

Request-scoped resources are created on demand for a request and then discarded afterwards. Critically for OSGi services the Jakarta RESTful Web Services whiteboard *must not* release a prototype scope service until *after* the response has completed. If the resource makes use of a Jakarta RESTful Web Services AsyncResponse, SseEventSink or a StreamingOutput then this may be some time after the return of resource method, and potentially on a different thread.

Jakarta RESTful Web Services whiteboard implementations must therefore take special care not to release request scoped instances until they are completely finished.

151.4.2.3 Asynchronous Responses

Jakarta RESTful Web Services supports asynchronous responses either for single-valued results, or for streams of data.

Single valued results may be provided by the AsyncResponse type which is injected into resource methods using the @Suspended annotation. If the resource is request scoped then the resource must not be released until after the AsyncResponse has completed.

The following example demonstrates the use of the AsyncResponse:

```
@Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 public class MyResource {

    @Path("foo")
    @GET
    public void getFoo(@Suspended AsyncResponse async) {
        Promise<String> p = doLongRunningTaskAsynchronously();
        p.onSuccess(v -> async.resume(v))
            .onFailure(t -> async.resume(t));
    }
}
```

Single valued asynchronous results can also be provided by returning a suitable type from the resource method. This can be a CompletionStage as described in the Jakarta RESTful Web Services specification, or an OSGi Promise type. In this case the response from the resource method will be sent once the returned type has completed, either successfully or by failing.

The following example demonstrates the use of an asynchronous return value:

```
@Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 public class MyResource {

    @Path("foo")
    @GET
    public Promise<String> getFoo() {
        Promise<String> p = doLongRunningTaskAsynchronously();
        return p;
    }
}
```

Multi-valued results in Jakarta RESTful Web Services are handled using Server Sent Events. To send Server Sent Events a Jakarta RESTful Web Services resource must declare its produced media type appropriately, and inject its resource method with a SseEventSink. The resource must also gain access to a Sse to use as a factory for Outbound Server Sent Events. If the resource is request scoped then the resource must not be released until after the SseEvent has closed.

The following example demonstrates the use of the Server Sent Events:

```
@Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 public class MyResource {

    @Context
    Sse sse;

    @GET
    @Produces(MediaType.SERVER_SENT_EVENTS)
    public void getFoo(@Context SseEventSink sink) {
        PushStream<String> p = getStreamOfMessages();
        p.map(sse::newEvent)
            .forEach(e -> sink::send)
            .onResolve(sink::close);
    }
}
```

151.4.3 Resource Service Properties

The following table describes the properties that can be used by Jakarta RESTful Web Services resources registered as Whiteboard services. Additionally, the common properties listed in Table 151.2 on page 1163 are supported.

Table 151.3 Service properties for Jakarta RESTful Web Services Whiteboard resource services.

Service Property	Type	Description
osgi.jakartars.resource JakartarsResource	String / Boolean *required*	Declares that this service must be processed by the Jakarta RESTful Web Services whiteboard when set to true. See JAKARTA_RS_RESOURCE.

151.4.4 A Jakarta RESTful Web Services Whiteboard Resource Example

The following example code uses Declarative Services annotations to register a Jakarta RESTful Web Services Whiteboard service.

```
@Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 public class MyResource {

     @GET
     @Path("hello")
     @Produces("text/plain")
     public String sayHello(){
          return "Hello World!";
     }
 }
```

This example registers the resource method at: /hello. Requests for http://www.acme.com/hello map to the resource method, which is called to process the request.

To associate the above example resource with another application add the following service property:

```
osgi.jakartars.application.select=(osgi.jakartars.name=myApp)
```

This can also be added using the property annotation:

```
@JakartarsApplicationSelect("(osgi.jakartars.name=myApp)")
```

Setting this property requires a Jakarta RESTful Web Services application named myApp to be registered:

```
@Component(service=Application.class)
@JakartarsName("myApp")
@JakartarsApplicationBase("foo")
public class MyApplication extends Application {}
```

Now the whiteboard resource will be available at http://www.acme.com/foo/hello as configured by the custom Jakarta RESTful Web Services application.

151.5 Registering Extensions

Jakarta RESTful Web Services extensions can be registered with the Jakarta RESTful Web Services Whiteboard by registering them as Whiteboard services. This means that the extension implementations are registered in the Service Registry. It is relatively common for a single extension type to provide more than one extension interface, for example MessageBodyReader and MessageBodyWriter are often provided by a single object.

Extension services must be registered with the Jakarta RESTful Web Services application that they target using only the interfaces that they advertise in the OSGi service registry. If, for example, an extension service object implements MessageBodyReader and ContainerRequestFilter but only advertises MessageBodyReader in its service registration then it must only be used as a MessageBodyReader

The following Jakarta RESTful Web Services extension interfaces are supported by this specification:

- ContainerRequestFilter and ContainerResponseFilter - these extensions are used to alter the HTTP request and response parameters.
- ReaderInterceptor and WriterInterceptor - these extensions are used to alter the incoming or outgoing objects for the call.
- MessageBodyReader and MessageBodyWriter - these extensions are used to deserialize/serialize objects to the wire for a given media type, for example application/json.
- ContextResolver extensions are used to provide objects for injection into other Jakarta RESTful Web Services resources and extensions.
- ExceptionMapper extensions are used to map exceptions thrown by Jakarta RESTful Web Services resources into responses.
- ParamConverterProvider extensions are used to map rich parameter types to and from String values.
- Feature and DynamicFeature - these extensions are used as a way to register multiple extension types with the Jakarta RESTful Web Services container. Dynamic Features further allow the extensions to be targeted to specific resources within the Jakarta RESTful Web Services container.

As Jakarta RESTful Web Services extensions have many possible interface types, none of which are defined by this specification, they must be registered with the osgi.jakartars.extension service property with a value of true. This property serves as a marker to the Jakarta RESTful Web Services whiteboard runtime, indicating that this OSGi service should be used as a Jakarta RESTful Web Services Whiteboard extension.

If the osgi.jakartars.extension is added to a service which does not advertise any of the Jakarta RESTful Web Services extension types then this is an error, and must result in a failure DTO being created.

151.5.1 Name Binding and Jakarta RESTful Web Services Extensions

By default Jakarta RESTful Web Services extensions are applied to every request, however sometimes they are only needed for a subset of resource methods. In this case a NameBinding annotation can be used to apply the extension to a subset of resource methods. The following example declares a binding annotation called FizzBuzz and uses it to bind an extension which replaces occurrences of "fizz" with "fizzbuzz".

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@NameBinding
public @interface FizzBuzz{}

@Component
@JakartarsExtension
@FizzBuzz
public class FizzBuzzReplacer implements WriterInterceptor {

    public void aroundWriteTo(WriterInterceptorContext ctx) {
        Object entity = ctx.getEntity();

        if(entity != null) {
            ctx.setEntity(entity.toString()
                .replace("fizz", "fizzbuzz"));
        }
        ctx.proceed();
    }
}

@Component(service=FizzResource.class)
@JakartarsResource
@Path("fizzbuzz")
public class FizzResource {

    @GET
    @FizzBuzz
    public String getFoos() {
        return "fizz, buzz, fizzbuzz";
    }
}
```

The result of an http request to the fizzbuzz URI will be *fizzbuzz, buzz, fizzbuzzbuzz*

The Jakarta RESTful Web Services whiteboard implementation must support the use of NameBinding to limit the scope of applied whiteboard extensions.

151.5.2 Extension ordering

Jakarta RESTful Web Services filters can be annotated with `@PreMatching` to indicate that they should be applied before the Jakarta RESTful Web Services container works out which resource should be called by the incoming request. These filters can therefore change the request such that it maps to a different resource than it would have before the filter's operation. Pre-matching filters cannot use `NameBinding` as no corresponding named resource is available to the runtime when they operate.

When used in the OSGi Jakarta RESTful Web Services Whiteboard Jakarta RESTful Web Services extensions follow the same ordering rules as defined by the Jakarta RESTful Web Services specification. Where more than one extension of a particular type is available then they are ordered according to their `jakarta.annotation.Priority`. If two extensions of the same type have the same priority then the whiteboard implementation must break the tie by ordering the extensions according to the natural ordering of their service references, as specified in `ServiceReference.compareTo`, with static extensions being ranked below all whiteboard services.

The extension processing flow is as follows:

1. Server receives a request
2. Pre-matching `ContainerRequestFilters` are executed. Changes made here can affect which resource method is chosen
3. The Server matches the request to a resource method
4. Post-matching ContainerRequestFilters are executed. This includes execution of all filters which match the incoming path and any name-bound filters.
5. ReaderInterceptors which match the incoming path are applied to the incoming request body. If the request has no body then the ReaderInterceptors are not called.
6. The list of `MessageBodyReaders` applicable to the path and incoming content type are tried according to the standard ordering rules. The first `MessageBodyReader` which states that it can deserialize the entity "wins" and is used to create the entity object. If the incoming request has no body then no `MessageBodyReaders` are called.
7. If the resource is request scoped then it is instantiated and injected with relevant types from any defined `ContextResolvers`. These are queried in order for each of the injectable fields.
8. The resource method is executed, passing any injected parameters from the request, and from any `ContextResolvers`. These are queried in turn for each of the injectable parameters.
9. `ContainerResponseFilters` are executed passing the method's response when it is complete. This includes execution of all filters, in order, which match the incoming path and any name-bound filters. Note that if an `AsyncResponse` is used then the response may not complete on the same thread as the incoming request.
10. `WriterInterceptors` which match the incoming path are applied to the outgoing response stream. If the response has no body then the WriterInterceptors are not called.
11. The list of `MessageBodyWriters` applicable to the path and outgoing content type are tried according to the standard ordering rules. The first writer which states that it can serialize the entity "wins" and is used to write out the entity object. If there is no response body then no writers are called.
12. The Server response is flushed and committed. If the resource that created the response was request scoped then it must only be released once the response is complete. Note that this may be at some point in the future, and on a different thread if the resource is using an `AsyncResponse`

151.5.3 Extension dependencies

The `osgi.jakartars.extension.select` property described in *Common Whiteboard Properties* on page 1163 applies to extensions as well as Jakarta RESTful Web Services resources. This is because one extension may depend on another.

The most common reason for an extension to have a dependency is for a context injection dependency. Dependencies are often provided by a ContextResolver so that they can be injected into another extension. The following example demonstrates a simple dependency on a Jackson ObjectMapper.

```
@JakartarsExtension
@JakartarsName("configProvider")
@Component
public class ConfigProvider implements ContextResolver {

    private ObjectMapper mapper = new ObjectMapper();

    public <T> getContext(Class<T> clazz) {
        if(ObjectMapper.class.equals(clazz)) {
            return mapper;
        }
        return null;
    }
}

@JakartarsExtension
@JakartarsExtensionSelect("(osgi.jakartars.name=configProvider)")
@Component(scope=ServiceScope.PROTOTYPE)
public class ConfiguredExtension implements WriterInterceptor {

    @Context
    private Providers providers;

    public void aroundWriteTo(WriterInterceptorContext ctx) {
        Object entity = ctx.getEntity();

        if(entity != null) {
            ObjectMapper mapper = providers
                    .getContextResolver(ObjectMapper.class)
                    .getContext(ObjectMapper.class);

            ctx.setEntity(mapper.writeValueAsString(entity));
        }
        ctx.proceed();
    }
}
```

151.5.4 Built in extensions

Depending on the capabilities of the Jakarta RESTful Web Services whiteboard implementation, and any statically defined extensions that make up a Jakarta RESTful Web Services Whiteboard application, there may be numerous non standard extensions available. These extensions must be represented using service properties on the Jakarta RESTful Web Services Service Runtime, or the whiteboard application as appropriate. This is why the extension select filters must also be matched against the Jakarta RESTful Web Services Service Runtime service and the whiteboard application being targeted.

151.5.5 Jakarta RESTful Web Services Whiteboard Extension Lifecycle

Jakarta RESTful Web Services extensions have a different lifecycle from Jakarta RESTful Web Services resources, within a single application a Jakarta RESTful Web Services extension always behaves as a singleton. If a Jakarta RESTful Web Services whiteboard extension is registered as prototype scope then the whiteboard implementation must obtain a separate instance for every application to which the extension is applied. Whiteboard extension services must be released by the Jakarta RESTful Web Services whiteboard when the application with which they have been registered is removed from the whiteboard, even if this is only a temporary situation.

Jakarta RESTful Web Services extensions often require configuration, and need to be configured differently for different applications. This configuration is typically provided by a Jakarta RESTful Web Services `ContextResolver` and injected into fields of the extension by the Jakarta RESTful Web Services container. It is therefore highly recommended that Jakarta RESTful Web Services Whiteboard extensions are always registered as prototype scope, so that separate instances can be created for each whiteboard application.

If an extension is registered as a singleton service then it should not rely on any fields being injected by the Jakarta RESTful Web Services Whiteboard implementation. Jakarta RESTful Web Services Whiteboard implementations may support field injection for singleton extensions, however this behavior is non portable, and may lead to errors at runtime when using other implementations.

151.5.6 Extension Service Properties

The following table describes the properties that can be used by Jakarta RESTful Web Services extensions registered as Whiteboard services. Additionally, the common properties listed in Table 151.2 on page 1163 are supported.

Table 151.4 Service properties for Jakarta RESTful Web Services Whiteboard extension services.

Service Property	Type	Description
osgi.jakartars.extension JakartarsExtension	String / Boolean *required*	Declares that this service must be processed by the Jakarta RESTful Web Services whiteboard when set to true. See JAKARTA_RS_EXTENSION.

151.5.7 A Jakarta RESTful Web Services Whiteboard Extension Example

The following example code uses Declarative Services annotations to register a require Jakarta RESTful Web Services Whiteboard extension which provides JSON support, and requires the extension from a Jakarta RESTful Web Services whiteboard resource.

```
@Component(property="serialize.to=JSON")
@JakartarsExtension
public class JsonProvider implements MessageBodyReader,
      MessageBodyWriter {
    ...
}

 @Component(service = Object.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 @JakartarsExtensionSelect("(serialize.to=JSON)")
 public class MyResource {

     @GET
     @Path("hello")
     @Produces(MediaType.APPLICATION_JSON)
     public List<String> getList(){
          return Arrays.asList("Hello", "World!");
     }
}
```

151.6 Registering RESTful Web Service Applications

The Jakarta RESTful Web Services specification defines the concept of an Application. An application is an object which collects together one or more Jakarta RESTful Web Services resources and extensions, and provides them to the Jakarta RESTful Web Services container. These resources may be provided as pre-instantiated singletons, or as Class objects to be reflectively instantiated.

The OSGi Jakarta RESTful Web Services whiteboard supports direct registration of Applications for two reasons:

- To support the use of legacy Jakarta RESTful Web Services applications with the whiteboard
- To provide simple scoping of Jakarta RESTful Web Services resources and extensions within a whiteboard, in this scenario it can be desirable to register an otherwise empty Application. This application can then be targeted by whiteboard services using the osgi.jakartars.application.select property.

Applications can be registered with the Jakarta RESTful Web Services Whiteboard by registering them as Whiteboard services which advertise themselves using the Jakarta RESTful Web Services Application type. In addition the whiteboard services must provide a osgi.jakartars.application.base property. The value of this property is the URI path relative to the root whiteboard context at which the application will be registered. Note that the value of any ApplicationPath annotation will be applied by the container in addition to the osgi.jakartars.application.base.

Each registered Whiteboard Application service is provided as a separate application within the whiteboard, and is isolated from other applications, including the default application. Whiteboard applications may be empty, may include zero or more static resources, and may include zero or more static extensions.

151.6.1 Application shadowing

The base URI for each application within the whiteboard must be unique. If two or more applications targeting the same whiteboard are registered with the same base URI then only the first in ranking order, as specified in ServiceReference.compareTo, will be made available. All other application services with that URI will have a failure DTO created for them. The same rules also apply to the osgi.jakartars.name property, with the highest ranked service shadowing other applications with the same name.

The default application is implicitly created by the whiteboard and has the name .default. The default application has a lower ranking than all registered services. Therefore an application registered with a base of / will shadow a default application bound at /.

A whiteboard application service may set an osgi.jakartars.name of .default to replace the default application. This technique may be used to rebind the default application to a base uri other than /.

If a whiteboard application fails (for example if the service get fails), or cannot be immediately deployed (for example if it has an unsatisfied osgi.jakartars.extension.select) then any applications that it shadows are still shadowed and relevant failure DTOs are created for all of the applications.

151.6.2 Application Extension Dependencies

It is possible for an application to require additional whiteboard extensions before it is eligible to be hosted by the whiteboard. When making this determination the Whiteboard implementation must perform a dry-run validation of the osgi.jakartars.extension.select filter, applying all of the whiteboard extensions targeted to the application before determining whether the application's requirements are met.

151.6.3 Application Service Properties

The following table describes the properties that can be used by Jakarta RESTful Web Services applications registered as Whiteboard services. Additionally, the common properties listed in Table 151.2 on page 1163 are supported, except for the osgi.jakartars.application.select property.

Table 151.5 Service properties for Jakarta RESTful Web Services Whiteboard application services.

Service Property	Type	Description
osgi.jakartars.application.base JakartarsApplicationBase	String *required*	Declares that this service must be processed by the Jakarta RESTful Web Services whiteboard, and defines the URI, relative to the root context of the whiteboard, at which the Application should be registered. See JAKARTA_RS_APPLICATION_BASE.

151.6.4 Accessing the Application service properties

In Jakarta RESTful Web Services the @Context annotation may be used to inject the Application instance into a resource or extension. Application configuration properties can also be injected using the Configuration type.

When using the Jakarta RESTful Web Services Whiteboard it can also be necessary to access the service properties associated with the application hosting the resource, for example to allow customization of the resource's response. To this end, the Jakarta RESTful Web Services whiteboard implementation must make the Application service properties available as a Map in the configuration. The key used to store this map is osgi.jakartars.application.serviceProperties, and it can be found in any injected Configuration instance.

Furthermore, for Feature and DynamicFeature extensions the application service properties must be visible in the FeatureContext passed to the extension when applying it to the application. The FeatureContext interface provides programmatic access to the Configuration for the application, so this visibility is achieved in the same manner as for an injected Configuration instance.

In the case where the hosting application is not an OSGi service, for example a Whiteboard implementation may choose to provide its default application as an internal detail, then the osgi.jakartars.application.serviceProperties map must exist containing the osgi.jakartars.name of the application and the service properties associated with the JakartarsServiceRuntime service.

151.6.5 A Jakarta RESTful Web Services Whiteboard Application Example

The following example code uses Declarative Services annotations to register a Jakarta RESTful Web Services Whiteboard application, and shows how to target an additional whiteboard resource to that application.

```
@Component(service=Application.class)
@JakartarsApplicationBase("example")
@JakartarsName("myApp")
public class MyApplication extends Application {
    public Set<Class<?>> getClasses() {
        return new HashSet<>(Arrays.asList(StaticResource.class));
    }
}

 @Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 @JakartarsApplicationSelect("(osgi.jakartars.name=myApp)")
 public class MyResource {

     @GET
     @Path("hello")
     @Produces("text/plain")
     public List<String> getList(){
          return Arrays.asList("Hello", "World!");
     }
 }
```

The MyResource service will be available at http://www.acme.com/example/hello

151.7 Whiteboard Error Handling

There are a number of error cases where the Jakarta RESTful Web Services whiteboard may be unable to correctly register a resource. All of these cases must result in a failure DTO being created with the appropriate error code.

- *Failure to obtain a service instance* - In the case where a published service is unable to be obtained by the Jakarta RESTful Web Services whiteboard then the service is deny listed by the container. A failure DTO is made available from the JakartarsServiceRuntime representing the deny listed service object.
- *Invalid service objects* - Jakarta RESTful Web Services extension and Application objects are required to advertise certain interfaces, or to extend certain types. If a service advertises itself using a Jakarta RESTful Web Services whiteboard service property, but fails to advertise an appropriate Jakarta RESTful Web Services type, or fails to provide any resource methods then this is an error and the service must be deny listed by the container. A failure DTO is available from the JakartarsServiceRuntime representing the deny listed service object.
- *Overlapping Application mappings* - As with resources in a single application it is possible that two Jakarta RESTful Web Services resources will register for the same path across applications. In this case the application with the longer base URI is shadowed, and a failure DTO is available from the JakartarsServiceRuntime representing the shadowed Application. Note that determining when two Jakarta RESTful Web Services applications overlap requires an analysis of the resource paths and all of sub-resource paths. If any of these paths clash then the entirety of the shadowed application must be unregistered and marked as a failure. It is an implementation error for some application resource paths to be available while others are shadowed.
- *Class-Space Compatibility* - Much of the Jakarta RESTful Web Services mapping definition is handled using annotations with runtime visibility. As Jakarta RESTful Web Services beans are POJOs there is no guarantee of class-space compatibility when the Jakarta RESTful Web Services implementation searches for whiteboard services. The Jakarta RESTful Web Services whiteboard must therefore confirm that the registered service shares the correct view of the Jakarta RESTful Web Services packages. If the class space is not consistent then the Jakarta RESTful Web Services whiteboard container must not register the services, but instead should create a failure DTO indicating that the Jakarta RESTful Web Services object is unable to be registered due to an incompatible class-space.
- *Missing Required Extensions* - If a Jakarta RESTful Web Services resource or extension requires one or more extensions using a osgi.jakartars.extension.select filter then at any given time it is possible that the Jakarta RESTful Web Services container will not be able to host the resource. At this time a failure DTO must be created for the relevant resource or extension service.

151.8 The Jakarta RESTful Web Services Client API

The Jakarta RESTful Web Services specification includes a client API for making REST requests. The normal mechanism for obtaining a Client is to use a ClientBuilder, which is instantiated using a static factory method. Static factory methods require the reflective loading of classes and suffer from significant lifecycle issues, as there is no way to force indirectly wired objects to be discarded if the implementation bundle is stopped or uninstalled.

Jakarta RESTful Web Services implementations must therefore register their ClientBuilder implementations as OSGi services for bundles to use in making Client instances. The ClientBuilder must be registered as a prototype scoped service. This allows bundles to configure multiple separate Client instances, and ensures that separate bundles will never accidentally provide conflicting configuration to the same ClientBuilder instance.

151.8.1 Client Filters, Interceptors, Readers and Writers

While Container extensions can be made available using whiteboard services, the same is not true for Clients. There are two main reasons for this:

1. There is no simple way to scope the filters and interceptors that would be applied to a given client. In a multi-tenant environment this could lead to unexpected behaviors.
2. Clients are not, in general, expected to be extended by third parties. The Client model is designed to be used by a bundle when making requests from a REST API. If further requests need to be made by a different bundle then it should create and configure a separate client. This is different from the whiteboard server, where one container port may host several distinct sets of resources.

In order to add filters, interceptors, readers and writers to the Jakarta RESTful Web Services client users should use the `ClientBuilder#register()` method when building their client.

151.8.2 Reactive Clients

The Jakarta RESTful Web Services client API supports both synchronous and asynchronous calls. In Jakarta RESTful Web Services 2.1 the asynchronous behavior of the client was extended using the `RxInvoker` (reactive invoker) interface. All clients are required to support a reactive invoker which returns CompletionStage instances, however in OSGi the common representation of an asynchronous return is the `Promise`. This specification therefore provides the `PromiseRxInvoker` interface which can be used to obtain Promises from the Jakarta RESTful Web Services client.

It is the responsibility of the Jakarta RESTful Web Services whiteboard implementation to create instances of `PromiseRxInvoker`. The exact mechanism by which instances are created is undefined, however it is possible to register a portable factory to create `PromiseRxInvoker` instances by implementing the `RxInvokerProvider` interface and registering this type with the Jakarta RESTful Web Services client. This portable implementation, however, is forced to use a blocking model by the underlying Jakarta RESTful Web Services API, and so implementations may choose to implement a more optimized non-blocking model using internal types.

Clients of this specification may make use of the `PromiseRxInvoker` using normal Jakarta RESTful Web Services idioms. For example:

```
Client client = clientBuilder.build();
Promise<String> p = client.target(REST_SERVICE_URL)
    .path("/foo")
    .path("/{name}")
    .resolveTemplate("name", buzz)
    .request()
    .rx(PromiseRxInvoker.class)
    .get(String.class);
```

151.8.3 Consuming Server Sent Events

In Jakarta RESTful Web Services 2.1 support was added for Server Sent Events. These events are consumed by a REST client using the `SseEventSource`. The `SseEventSource` is not created by a Jakarta RESTful Web Services client instance, but is normally created using a static factory method, which does not work in a modular environment. Therefore the Jakarta RESTful Web Services whiteboard implementation must register a `SseEventSourceFactory` service in the service registry. This object serves as a factory for the Jakarta RESTful Web Services SSE types.

Note that the SseEventSource has no way to register filters or message body processors. All of the filters and necessary processors must be registered with the Jakarta RESTful Web Services client that is used to create the `WebTarget` used when building the SseEventSource. A client may therefore consume Server Sent Events in the following way:

```
Client client = clientBuilder.build();

WebTarget target = client.target(REST_SERVICE_URL)
    .path("/foo")
    .path("/{name}")
    .resolveTemplate("name", buzz);

SseEventSource source = sseFactory.newSource(target);

source.register(event -> doSomething(event));

source.open();
```

A `SseEventSource` may easily be converted into a `PushEventSource` (and consequently a `PushStream`) as follows. Note that the implementation does not respond to back-pressure requests and should typically be used with a buffer.

```
SseEventSource source = sseBuilder.newSource(target);

PushEventSource<InboundSseEvent> pes = pec ->
        source.register(e -> {
                                try {
                                    if(pec.accept(PushEvent.data(e)) < 0) {
                                        source.close();
                                    }
                                } catch (Exception e) {
                                    try {
                                        pec.accept(PushEvent.error(e));
                                    } finally {
                                        source.close();
                                    }
                                }
                            },
                        t -> pec.accept(PushEvent.error(t)),
                        () -> pec.accept(PushEvent.close()));
        source.open();
        return source;
    };
```

151.9 Portability and Interoperability

The extensions defined by the Jakarta RESTful Web Services specification make Jakarta RESTful Web Services runtimes highly plugable, and it is common to extend the behavior of an application using this model. In many cases the custom behaviors are specific to a particular use case, for example mapping a specific exception into a `Response`, and there is no need for portability. In some common cases, however, there are extensions that can be used across a great many applications.

In order to ensure that a Jakarta RESTful Web Services whiteboard application can make use of a common extension service in a portable way this specification defines standard service property names that should be registered, as appropriate, by whiteboard extension services, whiteboard applications with static extensions, and Jakarta RESTful Web Services whiteboard implementations that provide built-in extension capabilities.

151.9.1 Media Type support

A common use of the Jakarta RESTful Web Services extension mechanism is to provide support for additional media types, both for consuming incoming requests and for producing responses. All Jakarta RESTful Web Services whiteboards must implicitly support `text/plain` and `application/xml` (using JAXB), however commonly used media types, such as `application/json` must be provided as an extension.

To ensure that whiteboard resources can depend on support for a particular media type in a portable way this specification defines the `osgi.jakartars.media.type` property. This property key should be registered with one or more media types that are supported, and may be provided by:

- A Whiteboard extension - if the extension provides general purpose support for reading from and writing to a media type then it should register this property.
- A Whiteboard application - if the application provides general purpose support for reading from and writing to a media type using a static extension then it should register this property.
- A Jakarta RESTful Web Services Whiteboard implementation - if the implementation provides general purpose built-in support for reading from and writing to a media type then it should register this property. If the built-in extension is always available then it should also be advertised by the *osgi.service Capability* on page 1181 for the JakartarsServiceRuntime.

The term general purpose is used to indicate that the media type support must not require implementation specific mapping metadata (for example annotations) and should, at a minimum, work with the OSGi scalar types and DTOs. The property key is available as a constant in `JAKARTA_RS_MEDIA_TYPE`.

151.9.1.1 Media Type names, wildcards and suffixes

Where possible the value(s) of the `osgi.jakartars.media.type` property should use the IANA registered names of the media type(s) supported, for example `application/json`. Officially registered media types are available from [4] *IANA Media Type Registrations*. If there is no officially registered media type then a vendor type should be used. Personal types may also be used, however due to the lack of portability afforded by personal types it is recommended that a non-standard property key is used for personal types.

Wildcard types (containing a `*`) are often used by extensions to indicate that they can create a variety of different media types. Rarely this is because the extension can serialize into multiple different formats. More typically this is because the extension can serialize into a format which has multiple names, or multiple formats which use the same basic serialization. Suffixes can further modify this behavior, for example VCards may be serialized as XML using `application/vcard+xml` or as JSON using `application/vcard+json`.

Wildcard types must not be used as values for the `osgi.jakartars.media.type` property as these do not provide sufficient information for whiteboard resources to reliably select a media type provider. Where a provider wishes to advertise support for a general suffix, for example `+json` or `+cbor` then the provider must advertise the primary media type associated with the suffix; in the supplied example these would be `application/json` and `application/cbor`. Clients wishing to use suffixed types should therefore also depend on the primary media type, not the suffixed type, if they wish to be portable. Where greater specificity is required it is recommended that the extension be selected based on additional custom properties. This should also be used for suffixes that have no primary type, for example `+der`. Official media type registrations are available from [5] *IANA Media Type Suffix Registrations*

151.9.1.2 Media Type Selection Example

The most commonly required media type for Jakarta RESTful Web Services services is application/json. To this end this specification defines a Component Property annotation JSONRequired which can be applied to a Declarative Services component to express:

- An extension requirement for runtime application/json media type support
- A requirement for the Jakarta RESTful Web Services whiteboard
- An optional active time requirement for application/json media type support, for use in application resolution/assembly.

Custom third-party annotations can easily be created to support additional media types as necessary, and are used as follows:

```
@Component(service = MyResource.class,
      scope = ServiceScope.PROTOTYPE)
 @JakartarsResource
 @JSONRequired
 @Produces(MediaType.APPLICATION_JSON)
 public class MyResource {

    @Path("foo")
    @GET
    public List<String> getFoos() {
        return Arrays.asList("foo", "bar", "baz");
    }
}
```

A corresponding component property type (JakartarsMediaType) exists for use on a Jakarta RESTful Web Services whiteboard extension or application service which provides media type support. This can be used to declare that one or more media types are supported.

```
@Component(scope = ServiceScope.PROTOTYPE)
 @JakartarsExtension
 @JakartarsMediaType(MediaType.APPLICATION_JSON)
 public class MyFeature implements Feature {

    public boolean configure(FeatureContext context) {
      context.register(MyJSONCodec.class);
        return true;
    }
}
```

151.10 Capabilities

151.10.1 osgi.implementation Capability

The Jakarta RESTful Web Services Whiteboard implementation bundle must provide the osgi.implementation capability with name osgi.jakartars. This capability can be used by provisioning tools and during resolution to ensure that a Jakarta RESTful Web Services Whiteboard implementation is present to process the Whiteboard services defined in this specification. The capability must also declare a uses constraint for the jakarta.ws.rs.* specification packages, and for the and OSGi Jakarta RESTful Web Services Whiteboard package. The version of this capability must match the version of this specification:

```
Provide-Capability: osgi.implementation;
```

```
        osgi.implementation="osgi.jakartars";
        uses:="jakarta.ws.rs, jakarta.ws.rs.client, jakarta.ws.rs.container,
               jakarta.ws.rs.core, jakarta.ws.rs.ext, jakarta.ws.rs.sse,
               org.osgi.service.jakartars.whiteboard";
        version:Version="2.0"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

151.10.2 osgi.contract Capability

The Jakarta RESTful Web Services Whiteboard implementation must provide a capability in the osgi.contract namespace with the name JakartaRESTfulWebServices if it exports the Jakarta RESTful Web Services specification packages. See [5] *Portable Java Contract Definitions*.

Providing the osgi.contract capability enables developer to build portable bundles for packages that are not versioned under OSGi Semantic Versioning rules. For more details see *osgi.contract Namespace* on page 709.

If the Jakarta RESTful Web Services API is provided by another bundle, the Jakarta RESTful Web Services Whiteboard implementation must be a consumer of the API and require the contract.

151.10.3 osgi.service Capability

The bundle providing the JakartarsServiceRuntime service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.jakartars.runtime and org.osgi.service.jakartars.runtime.dto packages:

```
Provide-Capability: osgi.service;
  objectClass:List<String>=
    "org.osgi.service.jakartars.runtime.JakartarsServiceRuntime";
  uses:="org.osgi.service.jakartars.runtime,
         org.osgi.service.jakartars.runtime.dto"
```

The bundle providing the jakarta.ws.rs.client.ClientBuilder service must also provide a capability in the osgi.service namespace representing this service. This capability must declare that the service is prototype scope, and that there is a uses constraint for the jakarta.ws.rs.client package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="jakarta.ws.rs.client.ClientBuilder";
  uses:="jakarta.ws.rs.client,org.osgi.service.jakartars.client";
  service.scope="prototype"
```

The bundle providing the org.osgi.service.jakartars.client.SseEventSourceFactory service must also provide a capability in the osgi.service namespace representing this service. This capability must declare a uses constraint for the org.osgi.service.jakartars.client package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>=
    "org.osgi.service.jakartars.client.SseEventSourceFactory";
  uses:="org.osgi.service.jakartars.client"
```

These capabilities must follow the rules defined for the *osgi.service Namespace* on page 711.

151.11 Security

This section only applies when executing in an OSGi environment which is enforcing Java permissions.

151.11.1 Service Permissions

Bundles that need to register Jakarta RESTful Web Services Whiteboard services must be granted ServicePermission[interfaceName, REGISTER] where interface name is the relevant Jakarta RESTful Web Services Whiteboard service interface name.

The Http Whiteboard implementation must be granted ServicePermission[*, GET] to retrieve the Jakarta RESTful Web Services Whiteboard services from the service registry.

151.11.2 Runtime Introspection

Bundles that need to introspect the state of the Jakarta RESTful Web Services runtime will need ServicePermission[org.osgi.service.jakartars.runtime.JakartarsServiceRuntime, GET] to obtain the Jakarta RESTful Web Services Service Runtime service and access the DTO types.

151.11.3 Calling Jakarta RESTful Web Services Whiteboard Services

This specification does not require that the Jakarta RESTful Web Services Whiteboard implementation is granted All Permission or wraps calls to the Jakarta RESTful Web Services Whiteboard services in a doPrivileged block. Therefore, it is the responsibility of the Jakarta RESTful Web Services Whiteboard services to use a doPrivileged block when performing privileged operations.

151.12 org.osgi.service.jakartars.client

Jakarta RESTful Web Services Whiteboard Client Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jakartars.client; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jakartars.client; version="[1.0,1.1)"
```

151.12.1 Summary

- PromiseRxInvoker - A specialization of the RxInvoker which creates Promise instances.
- SseEventSourceFactory - A factory for SseEventSource instances.

151.12.2 public interface PromiseRxInvoker extends RxInvoker<Promise>

A specialization of the RxInvoker which creates Promise instances.

Bundles may obtain an instance of a PromiseRxInvoker using a ClientBuilder obtained from the service registry and calling the jakarta.ws.rs.client.Invocation.Builder.rx(Class) method.

Provider Type Consumers of this API must not implement this type

151.12.2.1 public Promise<Response> delete()

151.12.2.2 public Promise<R> delete(Class<R> arg0)

Type Parameters <R>

151.12.2.3 **public Promise<R> delete(GenericType<R> arg0)**

Type Parameters <R>

151.12.2.4 **public Promise<Response> get()**

151.12.2.5 **public Promise<R> get(Class<R> arg0)**

Type Parameters <R>

151.12.2.6 **public Promise<R> get(GenericType<R> arg0)**

Type Parameters <R>

151.12.2.7 **public Promise<Response> head()**

151.12.2.8 **public Promise<R> method(String arg0, Class<R> arg1)**

Type Parameters <R>

151.12.2.9 **public Promise<R> method(String arg0, Entity<?> arg1, Class<R> arg2)**

Type Parameters <R>

151.12.2.10 **public Promise<R> method(String arg0, Entity<?> arg1, GenericType<R> arg2)**

Type Parameters <R>

151.12.2.11 **public Promise<Response> method(String arg0, Entity<?> arg1)**

151.12.2.12 **public Promise<R> method(String arg0, GenericType<R> arg1)**

Type Parameters <R>

151.12.2.13 **public Promise<Response> method(String arg0)**

151.12.2.14 **public Promise<Response> options()**

151.12.2.15 **public Promise<R> options(Class<R> arg0)**

Type Parameters <R>

151.12.2.16 **public Promise<R> options(GenericType<R> arg0)**

Type Parameters <R>

151.12.2.17 **public Promise<R> post(Entity<?> arg0, Class<R> arg1)**

Type Parameters <R>

151.12.2.18 **public Promise<R> post(Entity<?> arg0, GenericType<R> arg1)**

Type Parameters <R>

151.12.2.19 **public Promise<Response> post(Entity<?> arg0)**

151.12.2.20 **public Promise<R> put(Entity<?> arg0, Class<R> arg1)**

Type Parameters <R>

151.12.2.21 public Promise<R> put(Entity<?> arg0, GenericType<R> arg1)

Type Parameters <R>

151.12.2.22 public Promise<Response> put(Entity<?> arg0)

151.12.2.23 public Promise<Response> trace()

151.12.2.24 public Promise<R> trace(Class<R> arg0)

Type Parameters <R>

151.12.2.25 public Promise<R> trace(GenericType<R> arg0)

Type Parameters <R>

151.12.3 public interface SseEventSourceFactory

A factory for SseEventSource instances.

Bundles may obtain an instance of a SseEventSourceFactory using the service registry. This service may then be used to construct SseEventSource instances for the supplied WebTarget.

Provider Type Consumers of this API must not implement this type

151.12.3.1 public SseEventSource.Builder newBuilder(WebTarget target)

target The web target to consume events from

□ Create a new jakarta.ws.rs.sse.SseEventSource.Builder

Returns a builder which can be used to further configure the event source

151.12.3.2 public SseEventSource newSource(WebTarget target)

target The web target to consume events from

□ Create a new SseEventSource

Returns a configured event source

151.13 org.osgi.service.jakartars.runtime

Jakarta RESTful Web Services Runtime Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jakartars.runtime; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jakartars.runtime; version="[1.0,1.1)"
```

151.13.1 Summary

- JakartarsServiceRuntime - The JakartarsServiceRuntime service represents the runtime information of a Jakarta RESTful Web Services Whiteboard implementation.

- JakartarsServiceRuntimeConstants - Defines standard names for Jakarta RESTful Web Services Runtime Service constants.

151.13.2 public interface JakartarsServiceRuntime

The JakartarsServiceRuntime service represents the runtime information of a Jakarta RESTful Web Services Whiteboard implementation.

It provides access to DTOs representing the current state of the service.

The JakartarsServiceRuntime service must be registered with the JakartarsServiceRuntimeConstants.JAKARTA_RS_SERVICE_ENDPOINT service property.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

151.13.2.1 public RuntimeDTO getRuntimeDTO()

□ Return the runtime DTO representing the current state.

Returns The runtime DTO.

151.13.3 public final class JakartarsServiceRuntimeConstants

Defines standard names for Jakarta RESTful Web Services Runtime Service constants.

151.13.3.1 public static final String JAKARTA_RS_SERVICE_ENDPOINT = "osgi.jakartars.endpoint"

Jakarta RESTful Web Services Runtime Service service property specifying the endpoints upon which the Jakarta RESTful Web Services implementation is available.

An endpoint value is a URL or a relative path, to which the Jakarta RESTful Web Services Whiteboard implementation is listening. For example, http://192.168.1.10:8080/ or /myapp/. A relative path may be used if the scheme and authority parts of the URL are not known, e.g. if a bridged Http Whiteboard implementation is used. If the Jakarta RESTful Web Services Whiteboard implementation is serving the root context and neither scheme nor authority is known, the value of the property is "/". Both, a URL and a relative path, must end with a slash.

A Jakarta RESTful Web Services Whiteboard implementation can be listening on multiple endpoints.

The value of this service property must be of type String, String[], or Collection<String>.

151.14 org.osgi.service.jakartars.runtime.dto

Jakarta RESTful Web Services Runtime DTO Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jakartars.runtime.dto; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jakartars.runtime.dto; version="[1.0,1.1)"
```

151.14.1 Summary

- ApplicationDTO - Represents a Jakarta RESTful Web Services Application service.

- `BaseApplicationDTO` - Represents common information about a Jakarta RESTful Web Services application service.
- `BaseDTO` - Represents common information about a Jakarta RESTful Web Services service.
- `BaseExtensionDTO` - Represents common information about a Jakarta RESTful Web Services extension service.
- `DTOConstants` - Defines standard constants for the DTOs.
- `ExtensionDTO` - Represents a Jakarta RESTful Web Services Extension service currently being hosted by the JakartarsServiceRuntime
- `FailedApplicationDTO` - Represents a Jakarta RESTful Web Services service which is currently not being used due to a problem.
- `FailedExtensionDTO` - Represents a Jakarta RESTful Web Services Extension service which is currently not being used due to a problem.
- `FailedResourceDTO` - Represents a Jakarta RESTful Web Services resource service which is currently not being used due to a problem.
- `ResourceDTO` - Represents common information about a Jakarta RESTful Web Services resource service.
- `ResourceMethodInfoDTO` - Represents information about a Jakarta RESTful Web Services resource method.
- `RuntimeDTO` - Represents the state of a Jakarta RESTful Web Services Service Runtime.

151.14.2 public class ApplicationDTO extends BaseApplicationDTO

Represents a Jakarta RESTful Web Services Application service.

Concurrency Not Thread-safe

151.14.2.1 public ResourceMethodInfoDTO[] resourceMethods

The RequestPaths handled by statically defined resources in this Application

151.14.2.2 public ApplicationDTO()

151.14.3 public abstract class BaseApplicationDTO extends BaseDTO

Represents common information about a Jakarta RESTful Web Services application service.

Concurrency Not Thread-safe

151.14.3.1 public String base

The base URI of the resource defined by JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_BASE.

151.14.3.2 public ExtensionDTO[] extensionDTOs

Returns the representations of the dynamic Jakarta RESTful Web Services extension services associated with this Application. The returned array may be empty if this application is currently not associated with any Jakarta RESTful Web Services extension services.

151.14.3.3 public ResourceDTO[] resourceDTOs

Returns the representations of the dynamic Jakarta RESTful Web Services resource services associated with this Application. The returned array may be empty if this application is currently not associated with any Jakarta RESTful Web Services Resource services.

151.14.3.4 public BaseApplicationDTO()

151.14.4 public abstract class BaseDTO extends DTO

Represents common information about a Jakarta RESTful Web Services service.

Concurrency Not Thread-safe

151.14.4.1 public String name

The name of the service if it set one using JakartarsWhiteboardConstants.JAKARTA_RS_NAME, otherwise this value will contain the generated name for this service

151.14.4.2 public long serviceId

Service property identifying the Jakarta RESTful Web Services service

151.14.4.3 public BaseDTO()

151.14.5 public abstract class BaseExtensionDTO extends BaseDTO

Represents common information about a Jakarta RESTful Web Services extension service.

Concurrency Not Thread-safe

151.14.5.1 public String[] extensionTypes

The extension types recognized for this service.

151.14.5.2 public BaseExtensionDTO()

151.14.6 public final class DTOConstants

Defines standard constants for the DTOs. The error codes are defined to take the same values as used by the Http Service Whiteboard

151.14.6.1 public static final int FAILURE_REASON_DUPLICATE_NAME = 6

The service is registered in the service registry with the JakartarsWhiteboardConstants.JAKARTA_RS_NAME property and a service with that name already exists in the runtime

151.14.6.2 public static final int FAILURE_REASON_NOT_AN_EXTENSION_TYPE = 4

The extension service is registered in the service registry but the service is not registered using a recognized extension type

151.14.6.3 public static final int FAILURE_REASON_REQUIRED_APPLICATION_UNAVAILABLE = 7

The service is registered in the service registry with the JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_SELECT property and the filters is not matched by any running application.

151.14.6.4 public static final int FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE = 5

The service is registered in the service registry with the JakartarsWhiteboardConstants.JAKARTA_RS_EXTENSION_SELECT property and one or more of the filters is not matched.

151.14.6.5 public static final int FAILURE_REASON_SERVICE_NOT_GETTABLE = 2

The service is registered in the service registry but getting the service fails as it returns null.

151.14.6.6 public static final int FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE = 1

Service is shadowed by another service.

For example, another service with the same service properties but having a higher service ranking. See org.osgi.framework.ServiceReference.compareTo(Object).

151.14.6.7 public static final int FAILURE_REASON_UNKNOWN = 0

Failure reason is unknown.

151.14.6.8 public static final int FAILURE_REASON_VALIDATION_FAILED = 3

The service is registered in the service registry but the service properties are invalid.

151.14.7 public class ExtensionDTO extends BaseExtensionDTO

Represents a Jakarta RESTful Web Services Extension service currently being hosted by the JakartarsServiceRuntime

Concurrency Not Thread-safe

151.14.7.1 public String[] consumes

The media types consumed by this service, if provided in an Consumes annotation

151.14.7.2 public ResourceDTO[] filteredByName

The resourceDTOs that are mapped to this extension using a NameBinding annotation

151.14.7.3 public String[] nameBindings

The full names of the NameBinding annotations applied to this extension, if any

151.14.7.4 public String[] produces

The media types produced by this service, if provided in an Produces annotation

151.14.7.5 public ExtensionDTO()

151.14.8 public class FailedApplicationDTO extends BaseApplicationDTO

Represents a Jakarta RESTful Web Services service which is currently not being used due to a problem.

The service represented by this DTO is not used due to a failure, but the BaseApplicationDTO.extensionDTOs and BaseApplicationDTO.resourceDTOs may be non-empty if whiteboard services have been associated with this failed application.

Concurrency Not Thread-safe

151.14.8.1 public int failureReason

The reason why the resource represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_VALIDATION_FAILED,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE,
DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE

151.14.8.2 public FailedApplicationDTO()

151.14.9 public class FailedExtensionDTO extends BaseExtensionDTO

Represents a Jakarta RESTful Web Services Extension service which is currently not being used due to a problem.

Concurrency Not Thread-safe

151.14.9.1 public int failureReason

The reason why the extension represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_VALIDATION_FAILED,
DTOConstants.FAILURE_REASON_NOT_AN_EXTENSION_TYPE,
DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE

151.14.9.2 public FailedExtensionDTO()

151.14.10 public class FailedResourceDTO extends BaseDTO

Represents a Jakarta RESTful Web Services resource service which is currently not being used due to a problem.

Concurrency Not Thread-safe

151.14.10.1 public int failureReason

The reason why the resource represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_VALIDATION_FAILED,
DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE

151.14.10.2 public FailedResourceDTO()

151.14.11 public class ResourceDTO extends BaseDTO

Represents common information about a Jakarta RESTful Web Services resource service.

Concurrency Not Thread-safe

151.14.11.1 public ResourceMethodInfoDTO[] resourceMethods

The RequestPaths handled by this resource

151.14.11.2 public ResourceDTO()

151.14.12 public class ResourceMethodInfoDTO extends DTO

Represents information about a Jakarta RESTful Web Services resource method. All information is determined by reading the relevant annotations, from the Jakarta RESTful Web Services type and not interpreted further. Dynamic information, or information provided in other ways may not be represented in this DTO.

Concurrency Not Thread-safe

151.14.12.1 public String[] consumingMimeType

The mime-type(s) consumed by this resource method, null if Consumes is not defined

151.14.12.2 public String method

The HTTP verb being handled, for example GET, DELETE, PUT, POST, HEAD, OPTIONS, null if no HttpMethod is defined

151.14.12.3 public String[] nameBindings

The NameBinding annotations that apply to this resource method, if any

151.14.12.4 public String path

The path of this resource method. Placeholder information present in the URI pattern will not be interpreted and simply returned as defined.

151.14.12.5 public String[] producingMimeType

The mime-type(s) produced by this resource method, null if Produces is not defined

151.14.12.6 public ResourceMethodInfoDTO()

151.14.13 public class RuntimeDTO extends DTO

Represents the state of a Jakarta RESTful Web Services Service Runtime.

Concurrency Not Thread-safe

151.14.13.1 public ApplicationDTO[] applicationDTOs

Returns the representations of the Jakarta RESTful Web Services Application services associated with this Runtime. The returned array may be empty if this whiteboard is currently not associated with any Jakarta RESTful Web Services application services.

151.14.13.2 public ApplicationDTO defaultApplication

Returns the current state of the default application for this Runtime.

151.14.13.3 public FailedApplicationDTO[] failedApplicationDTOs

Returns the representations of the Jakarta RESTful Web Services extension services targeted to this runtime but currently not used due to some problem. The returned array may be empty.

151.14.13.4 public FailedExtensionDTO[] failedExtensionDTOs

Returns the representations of the Jakarta RESTful Web Services extension services targeted to this runtime but currently not used due to some problem. The returned array may be empty.

151.14.13.5 public FailedResourceDTO[] failedResourceDTOs

Returns the representations of the Jakarta RESTful Web Services resource services targeted to this runtime but currently not used due to some problem. The returned array may be empty.

151.14.13.6 public ServiceReferenceDTO serviceDTO

The DTO for the corresponding `JakartarsServiceRuntime`. This value is never `null`.

151.14.13.7 public RuntimeDTO()

151.15 org.osgi.service.jakartars.whiteboard

Jakarta RESTful Web Services Whiteboard Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jakartars.whiteboard; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jakartars.whiteboard; version="[1.0,1.1)"
```

151.15.1 Summary

- `JakartarsWhiteboardConstants` - Defines standard constants for the Jakarta RESTful Web Services Whiteboard services.

151.15.2 public final class JakartarsWhiteboardConstants

Defines standard constants for the Jakarta RESTful Web Services Whiteboard services.

151.15.2.1 public static final String JAKARTA_RS_APPLICATION_BASE = "osgi.jakartars.application.base"

Service property specifying the base URI mapping for a Jakarta RESTful Web Services application service.

The specified uri is used to determine whether a request should be mapped to the resource. Services without this service property are ignored.

The value of this service property must be of type `String`, and will have a "/" prepended if no "/" exists.

If two applications are registered with the same base uri then the lower ranked service is failed with a cause of DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

151.15.2.2 public static final String JAKARTA_RS_APPLICATION_SELECT = "osgi.jakartars.application.select"

Service property specifying the target application for a Jakarta RESTful Web Services resource or extension service.

The specified filter(s) is/are used to determine whether a resource should be included in a particular application. Services without this service property are bound to the default Application.

If a filter property is registered and no application running in the whiteboard matches the filter then the service will be failed with a cause of DTOConstants.FAILURE_REASON_REQUIRED_APPLICATION_UNAVAILABLE

The value of this service property must be of type `String+`, and each entry must be a valid OSGi filter.

151.15.2.3 public static final String JAKARTA_RS_APPLICATION_SERVICE_PROPERTIES = "osgi.jakartars.application.serviceProperties"

The property key which can be used to find the application service properties inside an injected Configuration

151.15.2.4 public static final String JAKARTA_RS_DEFAULT_APPLICATION = ".default"

The name of the default Jakarta RESTful Web Services application in every Whiteboard instance.

151.15.2.5 public static final String JAKARTA_RS_EXTENSION = "osgi.jakartars.extension"

Service property specifying that a Jakarta RESTful Web Services resource service should be processed by the whiteboard.

The value of this service property must be of type `String` or Boolean and set to "true" or `true`.

A service providing this property must be registered as one or more of the following types:

- MessageBodyReader
- MessageBodyWriter
- ContainerRequestFilter
- ContainerResponseFilter
- ReaderInterceptor
- WriterInterceptor
- ContextResolver
- ExceptionMapper
- ParamConverterProvider
- Feature
- DynamicFeature

If a service with this property does not match any of the defined types then it is registered as a failure DTO with the error code DTOConstants.FAILURE_REASON_NOT_AN_EXTENSION_TYPE,

151.15.2.6 public static final String JAKARTA_RS_EXTENSION_SELECT = "osgi.jakartars.extension.select"

A Service property specifying one or more target filters used to select the set of Jakarta RESTful Web Services extension services required to support this whiteboard service.

A Jakarta RESTful Web Services Whiteboard service may require one or more extensions to be available so that it can function. For example a resource which declares that it `@Produces("text/json")` requires a MessageBodyWriter which supports JSON to be available.

This service property provides a String+ set of LDAP filters which will be applied to the service properties of all extensions available in the Jakarta RESTful Web Services container. If all of the filters are satisfied then this service is eligible to be hosted by the Jakarta RESTful Web Services container.

This service property may be declared by any Jakarta RESTful Web Services whiteboard service, whether it is a resource, or an extension.

If this service property is not specified, then no extensions are required.

If one or more filter properties are registered and no suitable
extension(s) are available then the service will be failed with a cause of
DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE

The value of this service property must be of type `String` and be a valid filter string.

151.15.2.7 public static final String JAKARTA_RS_MEDIA_TYPE = "osgi.jakartars.media.type"

A service property specifying that a Jakarta RESTful Web Services extension service, application service, or whiteboard implementation provides support for reading from and writing to a specific media type.

The value of this property will be one or more media type identifiers, and where possible IANA registered names, such as `application/json` should be used. The value must not be a wildcard type. Support for multiple media types that use the same suffix should be supported by registering the media type associated with the suffix.

151.15.2.8 public static final String JAKARTA_RS_NAME = "osgi.jakartars.name"

Service property specifying the name of a Jakarta RESTful Web Services whiteboard service.

This name is provided as a property on the registered Endpoint service so that the URI for a particular Jakarta RESTful Web Services service can be identified. If this service property is not specified, then no Endpoint information will be registered for this resource.

Resource names must be unique among all services associated with a single Whiteboard implementation. If a clashing name is registered then the lower ranked service will be failed with a cause of DTOConstants.FAILURE_REASON_DUPLICATE_NAME

The value of this service property must be of type String.

151.15.2.9 public static final String JAKARTA_RS_RESOURCE = "osgi.jakartars.resource"

Service property specifying that a Jakarta RESTful Web Services resource should be processed by the whiteboard.

The value of this service property must be of type String or Boolean and set to "true" or true.

151.15.2.10 public static final String JAKARTA_RS_WHITEBOARD_IMPLEMENTATION = "osgi.jakartars"

The name of the implementation capability for the Whiteboard Specification for Jakarta RESTful Web Services.

151.15.2.11 public static final String JAKARTA_RS_WHITEBOARD_SPECIFICATION_VERSION = "2.0"

The version of the implementation capability for the Whiteboard Specification for Jakarta RESTful Web Services.

151.15.2.12 public static final String JAKARTA_RS_WHITEBOARD_TARGET = "osgi.jakartars.whiteboard.target"

Service property specifying the target filter to select the Jakarta RESTful Web Services Whiteboard implementation to process the service.

A Jakarta RESTful Web Services Whiteboard implementation can define any number of service properties which can be referenced by the target filter. The service properties should always include the osgi.jakartars.endpoint service property if the endpoint information is known.

If this service property is not specified, then all Jakarta RESTful Web Services Whiteboard implementations can process the service.

The value of this service property must be of type String and be a valid filter string.

151.16 org.osgi.service.jakartars.whiteboard.annotations

Jakarta RESTful Web Services Whiteboard Annotations Package Version 1.0.

This package contains annotations that can be used to require the Jakarta RESTful Web Services Whiteboard implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

151.16.1 Summary

- RequireJakartarsWhiteboard - This annotation can be used to require the Jakarta RESTful Web Services Whiteboard implementation.

151.16.2 @RequireJakartarsWhiteboard

This annotation can be used to require the Jakarta RESTful Web Services Whiteboard implementation. It can be used directly, or as a meta-annotation.

This annotation is applied to several of the Jakarta RESTful Web Services Whiteboard component property annotations meaning that it does not normally need to be applied to Declarative Services components which use the Jakarta RESTful Web Services Whiteboard.

Retention CLASS

Target TYPE, PACKAGE

151.17 org.osgi.service.jakartars.whiteboard.propertytypes

Jakarta RESTful Web Services Whiteboard Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.jakartars.whiteboard; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.jakartars.whiteboard; version="[1.0,1.1)"

151.17.1 Summary

- JakartarsApplicationBase - Component Property Type for the osgi.jakartars.application.base service property.
- JakartarsApplicationSelect - Component Property Type for the osgi.jakartars.application.select service property.
- JakartarsExtension - Component Property Type for the osgi.jakartars.extension service property.
- JakartarsExtensionSelect - Component Property Type for the osgi.jakartars.extension.select service property.
- JakartarsMediaType - Component Property Type for the osgi.jakartars.media.type service property.
- JakartarsName - Component Property Type for the osgi.jakartars.name service property.
- JakartarsResource - Component Property Type for the osgi.jakartars.resource service property.
- JakartarsWhiteboardTarget - Component Property Type for the osgi.jakartars.whiteboard.target service property.
- JSONRequired - Component Property Type for requiring JSON media type support using the JakartarsWhiteboardConstants.JAKARTA_RS_MEDIA_TYPE service property.

151.17.2 @JakartarsApplicationBase

Component Property Type for the osgi.jakartars.application.base service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard application service to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_BASE service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.2.1 String value

□ Service property providing a base context URI for a Jakarta RESTful Web Services whiteboard application.

Returns The base URI for this application.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_BASE

151.17.2.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.3 @JakartarsApplicationSelect

Component Property Type for the osgi.jakartars.application.select service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard resource or extension service to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_SELECT service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.3.1 String[] value

□ Service property providing a OSGi filter(s) identifying the application(s) to which this service should be bound.

Returns The filter(s) for selecting the application(s) to bind to.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_APPLICATION_SELECT

151.17.3.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.4 @JakartarsExtension

Component Property Type for the osgi.jakartars.extension service property.

This annotation can be used on a Jakarta RESTful Web Services extension service to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_EXTENSION service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.4.1 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.5 @JakartarsExtensionSelect

Component Property Type for the osgi.jakartars.extension.select service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard resource or extension to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_EXTENSION_SELECT service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.5.1 String[] value

□ Service property providing one or more OSGi filters identifying the extension(s) or application features which this service requires to work.

Returns The filters for selecting the extensions to require.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_EXTENSION_SELECT

151.17.5.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.6 @JakartarsMediaType

Component Property Type for the osgi.jakartars.media.type service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard extension or application to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_MEDIA_TYPE service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.6.1 String[] value

□ Service property identifying the name(s) of media types supported by this service.

Returns The Jakarta RESTful Web Services media types supported.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_MEDIA_TYPE

151.17.6.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.7 @JakartarsName

Component Property Type for the osgi.jakartars.name service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard service to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_NAME service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.7.1 String value

□ Service property identifying the name of a Jakarta RESTful Web Services service for processing by the whiteboard.

Returns The Jakarta RESTful Web Services whiteboard service name.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_NAME

151.17.7.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.8 @JakartarsResource

Component Property Type for the osgi.jakartars.resource service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard resource to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_RESOURCE service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.8.1 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.9 @JakartarsWhiteboardTarget

Component Property Type for the osgi.jakartars.whiteboard.target service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard resource or extension service to declare the value of the JakartarsWhiteboardConstants.JAKARTA_RS_WHITEBOARD_TARGET service property.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.9.1 String value

□ Service property providing an OSGi filter identifying the whiteboard(s) to which this service should be bound.

Returns The filter for selecting the whiteboards to bind to.

See Also JakartarsWhiteboardConstants.JAKARTA_RS_WHITEBOARD_TARGET

151.17.9.2 String PREFIX_ = "osgi."

Prefix for the property name. This value is prepended to each property name.

151.17.10 @JSONRequired

Component Property Type for requiring JSON media type support using the JakartarsWhiteboardConstants.JAKARTA_RS_MEDIA_TYPE service property.

This annotation can be used on a Jakarta RESTful Web Services whiteboard resource to declare require that JSON support is available before the resource becomes active. It also adds an optional Requirement for a service providing this media type to aid with provisioning.

See Also Component Property Types

Retention CLASS

Target TYPE

151.17.10.1 String osgi_jakartars_extension_select default "(osgi.jakartars.media.type=application/json)"

□ Provides an extension selection filter for an extension supporting the JSON media type

Returns A filter requiring an osgi.jakartars.media.type of application/json

151.17.10.2 String FILTER = "(osgi.jakartars.media.type=application/json)"

A filter requiring an osgi.jakartars.media.type of application/json

151.18 References

[1] *Jakarta RESTful Web Services 3.0 Specification*
https://jakarta.ee/specifications/restful-ws/3.0/

[2] *Portable Java Contract Definitions*
https://docs.osgi.org/reference/portable-java-contracts.html

[3] *Whiteboard Pattern*
https://docs.osgi.org/whitepaper/whiteboard-pattern/

[4] *IANA Media Type Registrations*
https://www.iana.org/assignments/media-types/media-types.xhtml

[5] *IANA Media Type Suffix Registrations*
https://www.iana.org/assignments/media-type-structured-suffix/media-type-structured-suffix.xhtml

[6] *JAX-RS Service Whiteboard Specification*
https://docs.osgi.org/specification/osgi.cmpn/8.0.0/service.jaxrs.html

151.19 Changes

- This specification is providing the same functionality as its predecessor, [6] *JAX-RS Service Whiteboard Specification*, with the switch from the JAX-RS API to the Jakarta RESTful Web Services API.

  Due to the naming change numerous constants have been renamed, including the service properties used in the specification. This allows the old and new specifications to coexist in the same framework.
- The Application Selection Filter property has changed from a single String to a String+, allowing multiple filters to be supplied.

152 CDI Integration Specification

Version 1.0

152.1 Introduction

Contexts and Dependency Injection ([1] *CDI*) is the standard *dependency injection* technology for Java. [2] *CDI 2.0* is the current version.

The CDI specification is a composition of the following high level features:

> - A well-defined life cycle for stateful objects bound to life cycle contexts, where the set of contexts is extensible
> - A sophisticated, typesafe dependency injection mechanism, including the ability to select dependencies at either development or deployment time, without verbose configuration
> - Support for Java EE modularity and the Java EE component architecture - the modular structure of a Java EE application is taken into account when resolving dependencies between Java EE components
> - Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page
> - The ability to decorate injected objects
> - The ability to associate interceptors to objects via typesafe interceptor bindings
> - An event notification model
> - A web conversation context in addition to the three standard web contexts defined by the Java Servlets specification
> - A Service Provider Interface (SPI) allowing portable extensions to integrate cleanly with the container
>
> —CDI

This specification describes how OSGi is integrated into the CDI programming model and the interaction with these features.

152.1.1 Essentials

- *Dependency Injection* - Provide an advanced dependency injection framework for bundles that can create and wire objects and services together into an application.
- *Extender Model* - Enable the configuration of components inside a bundle based on configuration data provided by the bundle developer. The life cycle of these components is controlled by the extender based on the extended bundle's state.
- *Unencumbered* - Does not require any special bundle activator or other code to be written inside the bundle in order to have components instantiated and configured.
- *Services* - Enable the usage of OSGi services as injected dependencies.
- *Configuration* - Enable the usage of Configuration Admin configuration objects as injected dependencies.

- *Dependencies* - Allow components to depend on configuration objects and services and to register services, with the full breadth of the OSGi capabilities.
- *Reactive* - It must be possible to react to changes in the external dependencies with different policies.
- *Introspection* - It must be possible to introspect the service components.
- *Business Logic* - A focus on writing business logic by using the features of CDI and reusable functionality provided by extensions.
- *Familiarity* - Familiar to Java developers knowledgeable in CDI.

152.1.2 Entities

- **CDI Entities**
  - *CDI* - Contexts and Dependency Injection 2.0.
  - *Bean* - A Java class that satisfies the criteria of a bean as defined in CDI and which provides contextual objects that define application state and/or logic.
  - *Producer* - A producer method or field acts as a source of objects to be injected. It is an alternative to beans.
  - *Contextual Instance* - The object instances produced by beans or producers within a given *context.*
  - *Context* - A Service Provider Interface (SPI) defining the life cycle for a set of contextual instances. The context also determines which contextual instances of beans are visible to the contextual instances of other beans.
  - *Scope* - A (CDI) scope identifies a particular *Context* implementation. All beans have a scope and are therefore bound to a particular context implementation. A scope is represented by an annotation type. Any contextual instances produced from the bean exist within a context identified by the scope.
  - *Injection Point* - A location in a contextual instance or producer which is the target for injection for a contextual instance.
  - *Qualifier* - An annotation used to define a quality used for matching. Qualifiers are applied to injection points, beans, producers (among other things). CDI finds beans matching an injection point's type then makes sure the qualifiers of the bean match all those on the injection point.
  - *Stereotype* - An annotation meta-annotated with `javax.enterprise.inject.Stereotype` used to define a recurring role by aggregating a CDI scope and various other aspects into a reusable unit.
  - *Decorators and Interceptors* - Actors that intercept certain method invocations of contextual instances.
  - *Portable Extension* - A portable extension uses the CDI SPI to provide additional and reusable functionality to a set of CDI beans.
  - *CDI Container* - For each CDI bundle, required portable extensions are loaded , metadata and bean classes are analyzed to create a bean injection graph. This process is encapsulated by a CDI container.
- **Entities defined by this specification**
  - *CDI Bundle* - An OSGi bundle containing CDI beans.
  - *CDI Extension Bundle* - A bundle providing one or more portable extensions.
  - *CDI Component Runtime (CCR)* - The actor that manages the CDI containers and their life cycle and allows introspection of CDI containers.
  - *Configuration Object* - Configuration Admin object which implements the `Configuration` interface and contains configuration data.

- *Factory Configuration Object* - A Configuration Object having a factory PID whose instances for which there can be 0 or N are under the control of Configuration Admin, all sharing the same factory PID.
- *Single Configuration Object* - A configuration object that has no factory PID and remains singularly independent from all other configuration objects.
- *Component* - A set of beans whose life cycle is derived from it's dependencies.
- *Dependency* - A configuration object or service upon which beans depend. These dependencies are dynamic in that their life cycle is independently controlled by other actors within the OSGi Framework and CCR must properly accommodate for this.
- *Configuration Template* - The static metadata describing a configuration object dependency.
- *Reference Template* - The static metadata describing a reference dependency.
- *Component Template* - The static definition of a *component* combining all the metadata defined by its beans, and its dependencies. The component template does not change between restarts of the CDI bundle.
- *Component Scope* - A (CDI) scope defined by this specification that represents the granular life cycle associated with a set of dependencies.
- *Component Instance* - A runtime instance of the component template which observes and reacts to the state of the OSGi Framework based on the metadata of the component template.
- *Container Component* - A component encompassing all beans in the CDI container **not** in the component scope. The container component results in a single component instance.
- *Single Component* - A component that encompasses beans that have the *Component Scope*, whose dependencies may include single configuration objects and services. A single component results in a single component instance.
- *Factory Component* - A component that encompassed beans having the *Component Scope*, that are driven by factory configuration objects and whose dependencies may include single configuration objects and services. A factory component results in any number of component instances, one for every factory configuration object.

152.1.3 Synopsis

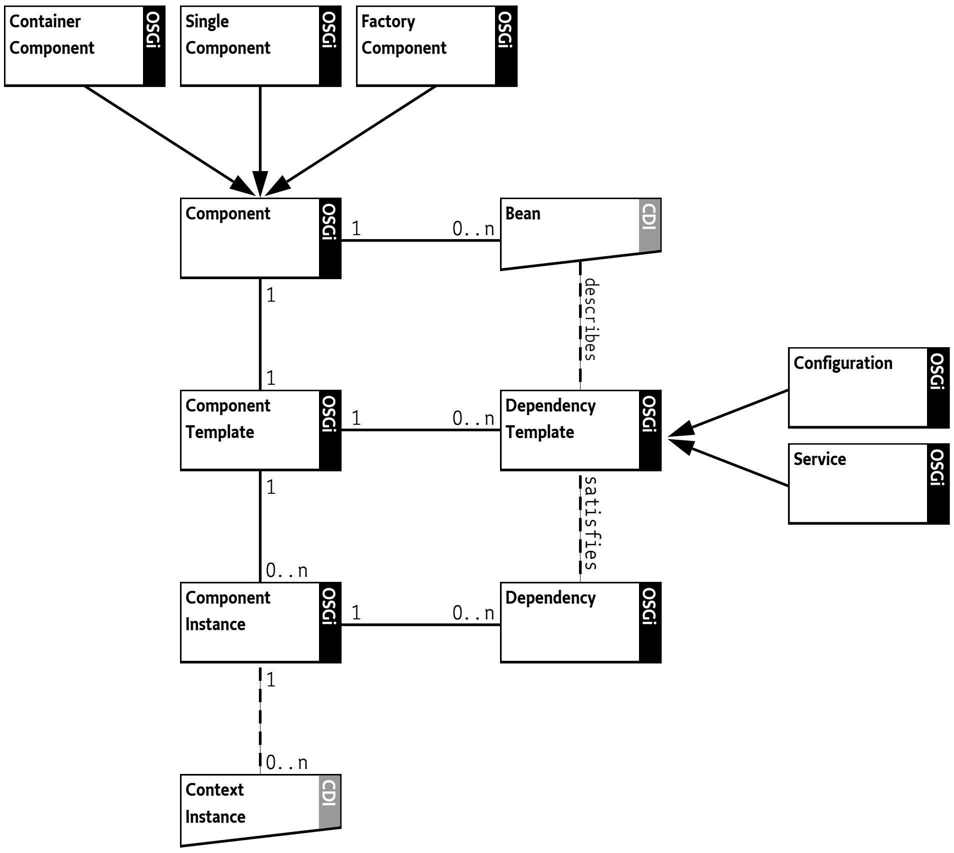
The CDI Extender reads CDI metadata from started CDI bundles. These metadata are in the form of XML documents, annotation types and requirements which define the set of *beans* available to the CDI container. Beans express dependencies on OSGi configuration objects and services and are assembled into components. The life cycle of a component is driven from the dependencies of its beans.

There are three types of components:

- *Container Component* - Consists of beans not in the *component scope*. There is exactly one container component per CDI bundle. It's life cycle is synonymous with the CDI container. The container component must be completely satisfied before other component types can be satisfied. The container component may provide multiple services. Altering the state of the container component's static dependencies results in the entire CDI container, and all other component types being destroyed and recreated.
- *Single Component* - A *Single Component* begins with a bean annotated by the @SingleComponent annotation and is further enhanced by other beans in it's injection graph that are component scoped. A single component may provide immediate functionality or a service resulting in an immediate instance or a single service registration. Unlike the container component, single components may be created, destroyed and react to changes in the state of it's dependencies in isolation, without affecting the entire CDI container. A single component's life cycle is driven first by the container component which must be satisfied and second by it's dependencies.
- *Factory Component* - A *Factory Component* begins with a bean annotated by the @FactoryComponent annotation and is further enhanced by other beans in it's injection graph that are component scoped. A factory component may provide immediate functionality or a service, resulting in

one immediate instance or service registration. Unlike the container component, factory components may be created, destroyed and react to changes in the state of it's dependencies in isolation, without affecting the entire CDI container. A factory component's life cycle is driven first by the container component which must be satisfied, secondly by factory configuration which result in one component instance per factory configuration object, and finally by it's dependencies.

Figure 152.1 *CCR Model*



152.2 Components

A traditional CDI application is composed of beans that have a well-defined life cycle based on the CDI scope they declare. This specification defines a component model in terms of beans and scopes as they are defined in the CDI specification in order to act as a good CDI citizen.

Components are defined by this specification to have the following characteristics:

- Components exist within a CDI bundle.
- Components are defined by collections of beans (referred to as component beans).
- Components may have dependencies on configuration objects and services. These dependencies are described using annotations defined by this specification.
- Components have properties, referred to as *component properties*. Some of these are defined by this specification and must be present. Others are aggregated from various configuration sources as defined in *Component Properties* on page 1211.

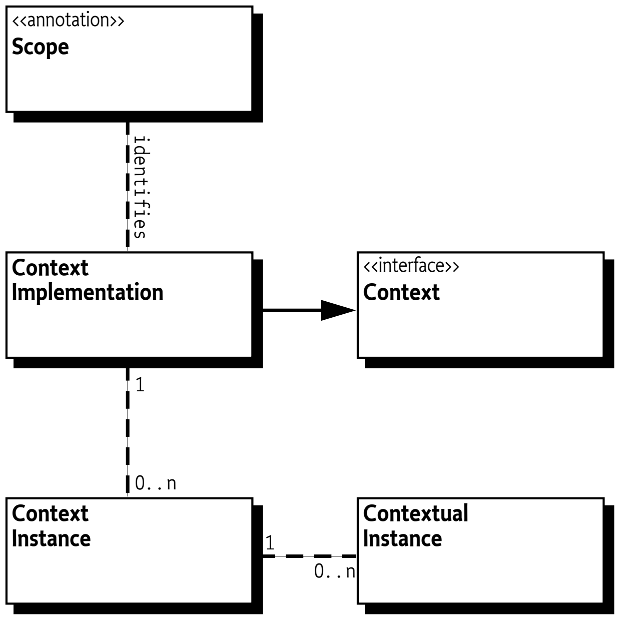
- Components have unique names within the CDI bundle.
- Components produce one or more *component instances*. Component instances are the runtime representation of the component. They independently react to the state of the dependencies declared by their component beans.

152.3 Component Scope

This specification uses the facilities of CDI [8] *Scopes and contexts* to define a life cycle for beans specifically for supporting a relationship with OSGi dependencies.

Associated with every CDI scope is an object implementing `javax.enterprise.context.spi.Context` or `javax.enterprise.context.spi.AlterableContext`. The life cycle and visibility rules for said scope are defined by this implementation which collaborates with the CDI container to create or destroy contextual instances. Contextual instances associated with the scope exist within a *context* which acts as a cache, creating new or returning existing contextual instances as needed. These contexts are managed by CCR in conjunction with the CDI container.

Figure 152.2 *CDI Scope Model*



The *component scope* is a [9] *Pseudo-scope* identified by the `@ComponentScoped` annotation. The *component scope* allows component instances to use component beans to create or destroy contextual instances when dependencies are satisfied or unsatisfied without interfering with the life cycle of other component instances (including the container component).

The context implementation must be registered with the CDI container using the CDI SPI. For example:

```
void afterBeanDiscovery(
 @Observes javax.enterprise.inject.spi.AfterBeanDiscovery abd) {

 Context ctx = ...
 abd.addContext(ctx);
}
```

The ComponentScoped annotation must be registered with the CDI container using the CDI SPI. For example:

```
void beforeBeanDiscovery(
 @Observes javax.enterprise.inject.spi.BeforeBeanDiscovery bbd) {

 bbd.addScope(ComponentScoped.class, false, false);
}
```

152.3.1 Contexts

The creation and destruction of the component scope's contexts must adhere to the following process:

- The following steps are taken to create a *context*:
  1. *the context is made active* - The method javax.enterprise.context.spi.Context.isActive() must return true.
  2. *contextual instances are created and injected* - Contextual instances can be retrieved by calling javax.enterprise.context.spi.Context.get(...).
  3. *the @Initialized event is fired* - On success of step 2, the CDI event @Initialized(ComponentScoped.class) is fired synchronously. See Table 152.1.

     When the component is a *single component*, the event payload is the contextual instance of the bean marked @SingleComponent.

     When the component is a *factory component* the event payload is the contextual instance of the bean marked @FactoryComponent.

     Any qualifiers defined on the bean of the contextual instance must be attached to the event.

     On failure of step 2, errors are logged and made available in errors.
  4. *the context is deactivated* - The method javax.enterprise.context.spi.Context.isActive() must return false.
- The following steps are taken to destroy a *context*:
  1. *the context is made active* - The method javax.enterprise.context.spi.Context.isActive() must return true.
  2. *the @BeforeDestroy is fired* - The CDI event @BeforeDestroy(ComponentScoped.class) is fired synchronously. See Table 152.1.

     When the component is a *single component* the event payload is the contextual instance of the bean marked @SingleComponent.

     When the component is a *factory component* the event payload is the contextual instance of the bean marked @FactoryComponent.

     Any qualifiers defined on the bean of the contextual instance must be attached to the event.
  3. *contextual instances are destroyed* - Any exceptions are logged.
  4. *the context is deactivated* - The method javax.enterprise.context.spi.Context.isActive() must return false.
  5. *the context is destroyed*
  6. *the @Destroyed event is fired* - The CDI event @Destroyed(ComponentScoped.class) is fired synchronously. See Table 152.1.

     When the component is a *single component* the event payload is the contextual instance of the bean marked @SingleComponent.

When the component is a *factory component* the event payload is the contextual instance of the bean marked @FactoryComponent.

Any qualifiers defined on the bean of the contextual instance must be attached to the event.

Note that the object may not be *usable* during this event because the context under which it was created is already destroyed.

Table 152.1 Component Context Events

Event Qualifier	Condition
@Initialized(ComponentScoped.class)	when a context is initialized and ready for use
@BeforeDestroy(ComponentScoped.class)	when a context is about to be destroyed, but before actual destruction
@Destroyed(ComponentScoped.class)	after a context is destroyed

152.3.1.1 When Contexts are Created

A *context* is created under each of the following conditions:

1. *Immediate instance* - A component instance that does not provide a service requires the immediate creation of a context.
2. *Singleton scoped service from a @SingleComponent* - A single component instance that provides a singleton scoped service requires the immediate creation of a context.

   The service object is the contextual instance of the bean marked @SingleComponent obtained from the context.
3. *Singleton scoped service from a @FactoryComponent* - A factory component instance that provides a singleton scoped service requires the immediate creation of a context for each factory configuration object.

   The service object is the contextual instance of the bean marked @FactoryComponent obtained from the context.
4. *Bundle scoped service* - A component instance that provides a bundle scope service requires the creation of a context when the ServiceFactory.getService() method is called.

   If the component is a *single component*, the service object is the contextual instance of the bean marked @SingleComponent obtained from the context.

   If the component is a *factory component*, the service object is the contextual instance of the bean marked @FactoryComponent obtained from the context.

   The context is released and destroyed when the ServiceFactory.ungetService() method is called.
5. *Prototyped scoped service* - A component instance that provides a prototype scope service requires the creation of a context when the PrototypeServiceFactory.getService() method is called.

   If the component is a *single component*, the service object is the contextual instance of the bean marked @SingleComponent obtained from the context.

   If the component is a *factory component*, the service object is the contextual instance of the bean marked @FactoryComponent obtained from the context.

   The context is released and destroyed when the PrototypeServiceFactory.ungetService() method is called.

In addition to the cases specified above, all contexts produced by an immediate component or by the service registration are released and destroyed when the component instance is no longer satisfied or when the CDI container is destroyed.

152.4 Container Component

The *container component* is composed of all the beans available to the CDI container which are **not** ComponentScoped.

The container component draws it's name from the CDI container id. By default, the CDI container id is equal to the Bundle-SymbolicName of the CDI bundle prefixed by 'osgi.cdi.'.

```
containerId ::= 'osgi.cdi.' bsn
bsn         ::= < Bundle-SymbolicName >
```

The container id can be specified using the container.id attribute of the CDI extender requirement in the bundle manifest. The value must follow the Bundle-SymbolicName syntax. For example:

```
Require-Capability:
 osgi.extender;
  filter:=”(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0.0)))”;
  container.id="my.id"
```

152.4.1 Container Component Configuration

The container component must be configurable using it's container id as a PID; referred to as the *container PID*.

```
containerPID ::= < container id >
```

Given a bundle with Bundle-SymbolicName equal to com.acme.bar which does not set the container.id attribute in the requirement, the container id would be:

```
osgi.cdi.com.acme.bar
```

From the requirement example above where the container id is set to my.id, the container PID would be:

```
my.id
```

The configuration object used to satisfy the container PID must be a single configuration object. However the configuration policy for this configuration object is *optional* and is not required to satisfy the container component.

152.4.2 Container Component Life Cycle

The container component is largely synonymous with the CDI container. When the dependencies of the container component are satisfied the CDI container completes it's initialization process and subsequently is fully functional. When the dependencies of the container component are no longer satisfied the CDI container is shutdown and all contextual instances are destroyed.

A container component with no beans would be immediately satisfied since it specifies no dependencies.

152.5 Standard Definitions

152.5.1 Annotation Inheritance

Annotations are not inherited unless meta-annotated by @java.lang.annotation.Inherited.

152.5.2 Code Examples

This specification provides several source code examples. In order to avoid repetition the following Java types are defined and re-used throughout:

```
interface Dog {}

interface Hound extends Dog {}

abstract class BassetHound implements Hound {}

class Spot extends BassetHound {}

class Buddy implements Hound {}
```

152.6 Single Component

A *Single Component* begins with and is rooted by a bean annotated by the @SingleComponent annotation. It is further enhanced by beans in it's injection graph that are `@ComponentScoped` which are discovered according to CDI's rules for [7] *Typesafe Resolution* starting from the `@SingleComponent` bean and recursing through all injection points until all injection points are resolved.

Resolution results which contain non-root beans marked with `@SingleComponent` or `@FactoryComponent` result in a definition error.

Any failed resolutions result in a definition error.

Applying any scope besides `@ComponentScoped` to a bean marked with `@SingleComponent` results in a definition error.

Any `@ComponentProperties` or `@Reference` injection point that is resolved by beans which are not provided by CCR results in a definition error.

A single component has an implicit dependency on the container component. Therefore it may never be satisfied until the container component is satisfied.

152.6.1 Single Component Naming

The `@SingleComponent` annotation is a *stereotype* which carries the `@javax.inject.Named` meta-annotation. This indicates that the default component name is:

> "the unqualified class name of the bean class, after converting the first character to lower case"
>
> —CDI

For example:

```
// component.name = fido
@SingleComponent
class Fido {}
```

However, the name may be specified by adding `@javax.inject.Named` directly to the bean and specifying a value whose syntax follows `cname` defined by the [11] *General Syntax Definitions*.

```
// component.name = Champ
@SingleComponent
@Named("Champ")
class Fido {}
```

152.6.2 Single Component Configuration

By default a single component must be configurable by using it's component name, prefixed by the container PID and a period (.), as a configuration PID. This *component PID* will be represented throughout the remained of the specification by the symbol Φ (capital Phi).

```
Φ            ::= containerPID '.' compName
containerPID ::= < container PID >
compName     ::= < component name >
```

A single component may change or add additional PIDs on which it depends. When multiple PIDs are referenced the order is relevant and affects the aggregation of the configuration objects into a flattened dictionary of component properties. Later PIDs take precedence over earlier PIDs. Also, it must be possible to reposition the component PID within the order. The PID annotation is used to control both referenced PIDs and their order.

The following is an example of a component that is configurable by it's component PID:

```
// component pids = [Φ]
@SingleComponent
class Fido {}
```

An example of a component replacing it's component PID with a specific PID:

```
// component pids = [com.acme.foo]
@SingleComponent
@PID("com.acme.foo")
class Fido {}
```

An example of multiple PIDs:

```
// component pids = [com.acme.foo, com.gamma.bar]
@SingleComponent
@PID("com.acme.foo")
@PID("com.gamma.bar")
class Fido {}
```

See *Component Properties* on page 1211 for how multiple component PIDs are merged into component properties.

Using @PID without arguments refers to the component PID:

```
// component pids = [Φ]
@SingleComponent
@PID
class Fido {}
```

This allows the component PID to be included anywhere in the order:

```
// component pids = [com.acme.foo, Φ, com.gamma.bar]
@SingleComponent
@PID("com.acme.foo")
@PID
@PID("com.gamma.bar")
class Fido {}
```

Each @PID annotation may specify a policy for the configuration object. The property policy is used to specify the value. The possible values are:

- *OPTIONAL* - A configuration object is not required. *This is the default policy.*

- *REQUIRED* - A configuration object is required.

```
// component pids = [com.acme.foo, Φ, com.gamma.bar]
@SingleComponent
@PID(value = "com.acme.foo", policy = Policy.REQUIRED)
@PID
@PID("com.gamma.bar")
class Fido {}
```

It is a definition error to refer to the same PID more than once.

The configuration objects used to satisfy the single component's referenced PIDs must be single configuration objects.

152.7 Factory Component

A *Factory Component* begins with and is rooted by a bean annotated by the @FactoryComponent annotation. It is further enhanced by beans in it's injection graph that are `@ComponentScoped` which are discovered according to CDI's rules for [7] *Typesafe Resolution* starting from the `@FactoryComponent` bean and recursing through all injection points until all injection points are resolved.

The `@FactoryComponent` annotation indicates that the component is bound to the life cycle of factory configuration objects associated with the factory PID specified in it's value property (or it's default component factory PID). Each factory configuration object associated with this factory PID results in a new *component instance*. The component properties of the component instance are supplemented by the properties of the factory configuration object.

Resolution results which contain a non-root bean marked with `@SingleComponent` or `@FactoryComponent` result in a definition error.

Any failed resolutions result in a definition error.

Applying any scope besides `@ComponentScoped` to a bean marked with `@FactoryComponent` results in a definition error.

Any `@ComponentProperties` or `@Reference` injection point that is resolved by beans which are not provided by CCR results in a definition error.

A factory component has an implicit dependency on the container component. Therefore it may never be satisfied until the container component is satisfied.

152.7.1 Factory Component Naming

The `@FactoryComponent` annotation is a *stereotype* which carries the `@javax.inject.Named` meta-annotation. This indicates that the default component name is:

> “the unqualified class name of the bean class, after converting the first character to lower case”
>
> —CDI

For example:

```
// component.name = fido
@FactoryComponent
class Fido {}
```

However, the name may be specified by adding `@javax.inject.Named` directly to the bean and specifying a value whose syntax follows `cname` defined by the [11] *General Syntax Definitions.*

```
// component.name = Champ
```

```
@FactoryComponent
@Named("Champ")
class Fido {}
```

152.7.2 Factory Component Configuration

By default a factory component must be configurable by using it's component name, prefixed by the container PID and a period (.), as a factory PID. This *component factory PID* will be represented throughout the remained of the specification by the symbol Σ (capital Sigma).

```
Σ            ::= containerPID '.' compName
containerPID ::= < container PID >
compName     ::= < component name >
```

An example of a factory component that is configurable by it's component factory PID:

```
// component pids = [Σ]
@FactoryComponent
class Fido {}
```

A factory component may specify a factory PID using it's value property. The value must conform to the syntax defined for the `Bundle-SymbolicName` header.

An example of a factory component specifying a factory PID:

```
// component pids = [com.acme.foo-####]
@FactoryComponent("com.acme.foo")
class Fido {}
```

A factory component may change or add additional PIDs on which it depends. When multiple PIDs are referenced the order is relevant and affects the aggregation of the configuration objects into a flattened dictionary of component properties. Later PIDs take precedence over earlier PIDs. The PID annotation is used to control both referenced PIDs and their order.

An example of multiple PIDs:

```
// component pids = [com.gamma.bar, com.acme.foo-####]
@FactoryComponent("com.acme.foo")
@PID("com.gamma.bar")
class Fido {}
```

Each `@PID` annotation may specify a policy for the configuration dependency. The property policy is used to specify the value. The possible values are:

- *OPTIONAL* - A configuration object is not required. *This is the default policy.*
- *REQUIRED* - A configuration object is required.

```
// component pids [com.acme.foo, com.gamma.bar, Σ]
@FactoryComponent
@PID(value = "com.acme.foo", policy = Policy.REQUIRED)
@PID("com.gamma.bar")
class Fido {}
```

See *Component Properties* on page 1211 for how multiple component PIDs are merged into component properties.

The component factory PID always reserves the highest precedence among specified PIDs and is positioned last in PID ordering for the purpose of aggregation

A factory component can only reference a single factory PID.

Notwithstanding the factory PID, it is a definition error to refer to the same PID more than once.

The configuration object used to satisfy the factory component's component factory PID must be a factory configuration object.

Configuration objects used to satisfy the PIDs referred to by the @PID annotations must be single configuration objects.

152.8 Component Properties

Each component instance is associated with a set of *component properties*. Component properties are specified in the following *configuration sources* (in order of precedence, where the properties provided by later lines overwrite those of earlier lines):

1. Properties specified as Bean Property Types on the bean annotated with @SingleComponent or @FactoryComponent must be treated according to *Bean Property Types* on page 1213.
2. Properties provided by single configuration objects whose PIDs are matched to and are processed in the order they are specified by the component.
3. Properties provided by a factory configuration object whose PID matches to the factory PID specified by the factory component.

The precedence behavior allows certain default values to be specified in component metadata while allowing properties to be replaced and extended by a configuration object.

Normally, a property value from a higher precedence configuration source replace a property value from a lower precedence configuration source. However, the service.pid property values receive different treatment. For the service.pid property, if the property appears multiple times in the configuration sources, CCR must aggregate all the values found into a Collection<String> having an iteration order such that the first item in the iteration is the property value from the lowest precedence configuration source and the last item in the iteration is the property value from the highest precedence configuration source. If the component refers to multiple PIDs, then the order of the service.pid property values collected from the corresponding configuration objects must match the order in which the PIDs are specified by the component. The values of the service.pid component property are the values as they come from the configuration sources and may container more values than those referred to by the component.

CCR always adds the following component properties, which cannot be overridden:

- component.name - The component name. The syntax for the component.name follows cname defined by the [11] *General Syntax Definitions*.
- component.id - A unique value (Long) that is larger than all previously assigned values. These values are not persistent across restarts of CCR.

152.8.1 Reference Properties

This specification defines some component properties which are associated with a specific reference. These are called *reference properties*. The name of a reference property for a reference is the name of the reference appended with a full stop ('.' \u002E) and a suffix unique to the reference property. Reference properties can be set wherever component properties can be set.

All component property names starting with a reference name followed by a full stop ('.' \u002E) are reserved for use by this specification.

Following are the reference properties defined by this specification.

152.8.1.1 Target Property

The *target property* is a reference property which aids in the selection of target services for the reference. See *Reference Injection Points* on page 1222. The name of a target property is the name of a reference appended with .target.

```
target  ::= refName '.target'
refName ::= < reference name >
```

For example, the target property for a reference with the name http

```
@Inject
@Reference
Http http;
```

would have the name http.target. The value of a target property is a filter String used to select target services for the reference.

```
http.target=(context.name=foo)
```

A default target property value can also be set by the @Reference.target property.

The target property value must be a valid filter String according to [12] *Filter Syntax*. Invalid filters result in unmatchable reference filters.

CCR must support the target property for all references.

152.8.1.2 Minimum Cardinality Property

The initial minimum cardinality of a reference is specified by the optionality of the reference. The minimum cardinality of a reference cannot exceed the multiplicity: a scalar reference has a multiplicity of 1 and a java.util.List or java.util.Collection reference has a multiplicity of n.

The *minimum cardinality property* is a reference property which can be used to raise the minimum cardinality of a reference from its initial value. That is, a 0..1 cardinality can be raised to a 1..1 cardinality by setting the reference's minimum cardinality property to 1. A 0..n cardinality can be raised to a m..n cardinality by setting the reference's minimum cardinality property to m such that m is a positive integer. The minimum cardinality of a reference cannot be lowered. A mandatory reference cannot be reduced to optional through this property. That is, a 1..1 cardinality can not be lowered to a 0..1 cardinality because the component was written to expect at least one bound service.

The name of a minimum cardinality property is the name of a reference appended with .cardinality.minimum.

```
minimumCardinality ::= refName '.cardinality.minimum'
refName            ::= < reference name >
```

For example, the minimum cardinality property for a reference with the name http

```
@Inject
@Reference
Http http;
```

would have the name http.cardinality.minimum.

```
http.cardinality.minimum=3
```

The value of a minimum cardinality property must be a positive integer or a value that can be coerced into a positive integer using the conversions defined by the *Converter Specification* on page 1457. If the numerical value of the minimum cardinality property is not valid for the reference's cardinality or the minimum cardinality property value cannot be coerced into a numerical value, then the minimum cardinality property must be ignored and a warning message logged.

Attempts to reduce the initial minimum cardinality will result in a warning message to be logged and the value to be otherwise ignored.

CCR must support the minimum cardinality property for all references.

152.9 Bean Property Types

Component properties can be defined and accessed through a user defined annotation type, called a *bean property type*, containing the property names, property types and default values. A bean property type allows properties to be defined and accessed in a type safe manner. Bean Property Types must be annotated with the BeanPropertyType meta-annotation.

The following example shows the definition of a bean property type called Props which defines three properties where the name of the property is the name of the method, the type of the property is the return type of the method and the default value for the property is the default value of the method.

```
@BeanPropertyType
public @interface Props {
 boolean enabled() default true;
 String[] names() default {"a", "b"};
 String topic() default "default/topic";
}
```

Bean Property Types can be used in several ways:

- Bean Property Types can be used along side the SingleComponent or FactoryComponent annotations to provide component properties.
- Bean Property Types can be used on ApplicationScoped or Dependent scoped beans, where the Service annotation is applied to provide service properties.
- Bean Property Types can be used on fields and methods annotated with @Produces, where the Service annotation is applied, to provide service properties.
- Bean Property Types can be used on injection points where the Reference annotation is applied, to provide target filter properties. Target filter properties can only provide AND filters.
- Bean Property Types can be used on injection points as the injection point type where the ComponentProperties annotation is applied to provide type safe coercion of component properties.

Each use defines property names, types and values.

The following example shows a component bean annotated with the example Props bean property type which specifies a property value for the component which is different than the default value. The example also shows an injection point method taking the example Props bean property type as the injection point type and the method implementation accesses component property values by invoking methods on the bean property type object.

```
@SingleComponent
@Props(names="myapp")
public class MyBean {
 @Inject
 void activate(Props props) {
  if (props.enabled()) {
   // do something
  }
  for (String name : props.names()) {
   // do something with each name
  }
 }
}
```

Bean Property Types must be defined as annotation types. This is done for several reasons. First, the limitations on annotation type definitions make them well suited for Bean Property Types. The methods must have no parameters and the return types supported are limited to a set which is well suited for component properties. Second, annotation types support default values which is useful for defining the default value of a component property. Finally, as annotations, they can be used to annotate bean classes.

At runtime, when CCR needs to provide injection points an object whose type is a bean property type, CCR must construct an instance of the bean property type whose methods are backed by the values of the component properties. This object can then be used to obtain the property values in a type safe manner.

152.9.1 Bean Property Type Mapping

Each method of a bean property type is mapped to a component property. The property name is derived from the method name. Certain common property name characters, such as full stop ('.' `\u002E`) and hyphen-minus ('-' `\u002D`) are not valid in Java identifiers. So the name of a method must be converted to its corresponding property name as follows:

- A single dollar sign ('$' `\u0024`) is removed unless it is followed by:
  - A low line ('_' `\u005F`) and a dollar sign in which case the three consecutive characters ("$_$") are converted to a single hyphen-minus ('-' `\u002D`).
  - Another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.
- A single low line ('_' `\u005F`) is converted into a full stop ('.' `\u002E`) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.
- If the bean property type declares a `PREFIX_` field whose value is a compile-time constant String, then the property name is prefixed with the value of the `PREFIX_` field.

Table 152.2 contains some name mapping examples.

Table 152.2 Bean Property Type Name Mapping Examples

Bean Property Type Method Name	Component Property Name
`myProperty143`	`myProperty143`
`$new`	`new`
`my$$prop`	`my$prop`
`dot_prop`	`dot.prop`
`_secret`	`.secret`
`another__prop`	`another_prop`
`three___prop`	`three_.prop`
`four_$__prop`	`four._prop`
`five_$_prop`	`five..prop`
`six$_$prop`	`six-prop`
`seven$$_$prop`	`seven$.prop`

However, if the bean property type is a *single-element annotation*, see 9.7.3 in [16] *The Java Language Specification, Java SE 8 Edition*, then the property name for the `value` method is derived from the name of the bean property type rather than the name of the method.

In this case, the simple name of the bean property type, that is, the name of the class without any package name or outer class name, if the bean property type is an inner class, must be converted to the property name as follows:

- When a lower case character is followed by an upper case character, a full stop ('.' \u002E) is inserted between them.
- Each upper case character is converted to lower case.
- All other characters are unchanged.
- If the bean property type declares a PREFIX_ field whose value is a compile-time constant String, then the property name is prefixed with the value of the PREFIX_ field.

Table 152.3 contains some mapping examples for the value method.

Table 152.3 *Single-Element Annotation Mapping Examples for* value *Method*

Bean Property Type Name	value **Method Component Property Name**
ServiceRanking	service.ranking
Some_Name	some_name
OSGiProperty	osgi.property

If the bean property type is a *marker annotation*, see 9.7.2 in [16] *The Java Language Specification, Java SE 8 Edition*, then the property name is derived from the name of the bean property type, as is described above for single-element annotations, and the value of the property is Boolean.TRUE. Marker annotations can be used to annotate component beans to set a component property to the value Boolean.TRUE. However, since marker annotations have no methods, they are of no use as injection point types.

The property type can be directly derived from the type of the method. All types supported for annotation elements can be used except for annotation types. Method types of an annotation type or array thereof are not supported.

If the method type is Class or Class[], then the property type must be String or String[], respectively, whose values are fully qualified class names in the form returned by the Class.getName() method.

If the method type is an enumeration type or an array thereof, then the property type must be String or String[], respectively, whose values are the names of the enum constants in the form returned by the Enum.name() method.

152.9.2 Coercing Bean Property Type Values

When a bean property type is used as an injection point type alone with @ComponentProperties, CCR must create a contextual instance that implements the bean property type and maps the methods of the bean property type to component properties. The name of the method is converted to the property name as described in *Bean Property Type Mapping* on page 1214. The property value may need to be coerced to the type of the method. In Table 152.4, the columns are source types, that is, the type of the component property value, and the rows are target types, that is, the method types. The property value is *v*; *number* is a primitive numerical type and *Number* is a wrapper numerical type. An invalid coercion is represented by throw. Such a coercion attempt must result in throwing a Bean Property Exception when the bean property type method is called. Any other coercion error, such as parsing a non-numerical String to a number or the inability to coerce a String into a Class or enum object, must be wrapped in a Bean Property Exception and thrown when the bean property type method is called.

Table 152.4 *Coercion From Property Value to Method Type*

target \ source	**String**	**Boolean**	**Character**	***Number***	**Collection/array**
String	*v*	*v*. toString()	*v*. toString()	*v*. toString()	If *v* has no elements, null; otherwise the first element of *v* is coerced.

target \ source	**String**	**Boolean**	**Character**	***Number***	**Collection/array**
boolean	Boolean. parseBoolean(*v*)	*v*. booleanValue()	*v*. charValue() != 0	*v*. doubleValue() != 0	If *v* has no elements, false; otherwise the first element of *v* is coerced.
char	*v*. length() > 0 ? *v*. charAt(0) : 0	*v*. booleanValue() ? 1 : 0	*v*. charValue()	(char) *v*. intValue()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
number	*Number*. parse*Number*(*v*)	*v*. booleanValue() ? 1 : 0	(*number*) *v*. charValue()	*v*. *number*Value()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
Class	Bundle. loadClass(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
EnumType	*EnumType*. valueOf(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
annotation type	throw	throw	throw	throw	throw
array	A single element array is created and *v* is coerced into the single element of the new array.				An array the size of *v* is created and each element of *v* is coerced into the corresponding element of the new array.

Component properties whose names do not map to bean property type methods are ignored. If there is no corresponding component property for a bean property type method, the bean property type method must:

- Return 0 for numerical and char method types.
- Return false for boolean method type.
- Return null for String, Class, and enum.
- Return an empty array for array method types.
- Throw a BeanPropertyException for annotation method types.

152.9.3 Standard Bean Property Types

Bean Property Types for standard service properties are specified in the org.osgi.service.cdi.propertytypes package.

The ServiceDescription bean property type can be used to add the service.description component property, service property or target filter. The ServiceRanking bean property type can be used to add the service.ranking component property, service property or target filter. The ServiceVendor bean property type can be used to add the service.vendor component property, service property or target filter. For example, using these Bean Property Types as annotations:

```
@FactoryComponent
@ServiceDescription(”My Acme Service implementation”)
@ServiceRanking(100)
@ServiceVendor("My Corp")
public class MyBean implements AcmeService {}
```

will result in the following component properties:

```
service.description=My Acme Service implementation # String
service.ranking=100 # Integer
service.vendor=My Corp # String
```

The ExportedService bean property type can be used to specify service properties for remote services.

152.10 Providing Services

A key aspect of working with OSGi is the ability to provide services. Services are published to the service registry specifying service types. The @Service annotation provides this capability to CCR and serves a dual role; the first of which is indicating that a bean publishes a service, the second indicating the service types. @Service can be applied in any one of the following ways:

152.10.1 @Service applied to bean class

Applying the @Service annotation to the bean class indicates the set of service types will be one of (in order of precedence):

1. *the specified type(s)* - When providing a specified value, these are the types under which the service is published.

   ```
   // service types = [BassetHound, Dog]
   @Service({BassetHound.class, Dog.class})
   class Spot {}
   ```

2. *directly implemented interfaces* - These are the interfaces for which the bean class directly specifies an implements clause.

   ```
   // service types = [Hound]
   @Service
   class Fido implements Hound {}
   ```

3. *bean class* - The class of the bean itself is the type under which the service is published.

   ```
   // service types = [Fido]
   @Service
   class Fido
   ```

The @Service annotation is *never inherited*. CCR ignores instances of the annotation on super classes, interfaces or super interfaces for this purpose.

152.10.2 @Service applied to type use

A convenient readability optimization is to apply the @Service annotation on *type_use*. This is to say that it may be applied to extends and/or implements clauses. For example:

```
// service types = [BassetHound]
class Fido extends @Service BassetHound {}
```

Or:

```
// service types = [Hound]
class Fido implements @Service Hound {}
```

The two approaches can be combined. @Service annotations are collected so that the service is published with all collected types:

```
// service types = [BassetHound, Hound]
class Fido extends @Service BassetHound implements @Service Hound {}
```

In this scenario, any use of the @Service.value property will result in a definition error.

Applying @Service to both bean class, and type use will result in a definition error.

152.10.3 @Service applied to Producers

Applying the @Service annotation to producer methods or fields indicates the set of service types as described in the following table (earlier rows take precedence over later rows).

Table 152.5 @Service applied to Producers

Case	Description

the type(s) specified by @Service.value	When providing a specified @Service.value, these are the types under which the service is published.
	`// service types = [BassetHound, Dog]`
	`@Produces`
	`@Service({BassetHound.class, Dog.class})`
	`Spot getSpot() {`
	` return new Spot();`
	`}`

the returned interface	In the case of a producer method, if the return type is an interface, this type is used as the service type.
	`// service types = [Dog]`
	`@Produces`
	`@Service`
	`Dog getDog() {`
	` return new Spot();`
	`}`

all directly implemented interfaces of returned type	In the case of a producer method, if the return type is a concrete type, use any interfaces directly implemented by the concrete type.
	`// service types = [Hound]`
	`@Produces`
	`@Service`
	`Buddy getBuddy() {`
	` return new Buddy();`
	`}`

the return type	In the case of a producer method, if the return type is a concrete type which does not directly implement any interfaces, use the concrete type.
	`class Fido {}`
	
	`// service types = [Fido]`
	`@Produces`
	`@Service`
	`Fido getFido() {`
	` return new Fido();`
	`}`

Case	Description

the field interface	In the case of a producer field, if the field type is an interface this type is used as the service type.
	`// service types = [Dog]`
	`@Produces`
	`@Service`
	`Dog dog = new Spot();`

all directly implemented interfaces of the field type	In the case of a producer field, if the field type is a concrete type use any interfaces directly implemented by the concrete type.
	`// service types = [Hound]`
	`@Produces`
	`@Service`
	`Buddy buddy = new Buddy();`

the field type	In the case of a producer field if the field type is a concrete type which does not directly implement any interfaces use the concrete type.
	`class Fido {}`
	
	`// service types = [Fido]`
	`@Produces`
	`@Service`
	`Fido fido = new Fido();`

152.10.4 @Service Type Restrictions

Regardless of the source, no service type may be a generic type. A generic type found in the set of service types will result in a definition error.

Service types must be a subset of bean types, including types restricted by the use of the @javax.enterprise.inject.Typed annotation. This restriction is required to support CDI features like *Decorators* and *Interceptors*.

Using the @Service annotation on injection points will result in a definition error.

152.10.5 Service Properties

The main source of service properties is *Component Properties* on page 1211.

When CCR registers a service on behalf of a component instance, CCR must follow the recommendations in *Property Propagation* on page 73 and must not propagate private configuration properties. That is, the service properties of the registered service must be all the component properties of the component configuration whose property names do not start with full stop ('.' \u002E).

Component properties whose names start with full stop are available to the component instance but are not available as service properties of the registered service.

152.10.5.1 Container component service properties

In addition to component properties, services provided by the container component obtain additional service properties from Bean Property Types on the bean or producer providing the service. See *Bean Property Types* on page 1213.

152.10.6 Service Scope

Service scope represents the scope of the registered service object. There are three scopes supported by the OSGi Framework. Each can be represented in CCR.

- *Bundle scope* - In order to specify a bundle scoped service, the @ServiceInstance annotation is specified on the bean class, producer method or producer field with the value BUNDLE.

  ```
  @Service
  @ServiceInstance(ServiceScope.BUNDLE)
  class Fido implements Hound {}
  ```

- *Prototype scope* - In order to specify a prototype scoped service, the @ServiceInstance annotation is specified on the bean class, producer method or producer field with the value PROTOTYPE. The service object is the contextual instance created by the producer or bean.

  ```
  @Service
  @ServiceInstance(ServiceScope.PROTOTYPE)
  class Fido implements Hound {}
  ```

- *Singleton scope* - Unless otherwise specified, services are singleton scoped but the scope can be explicitly expressed if the @ServiceInstance annotation is specified on the bean class, producer method or producer field with the value SINGLETON. The service object is the contextual instance created by the producer or bean.

  ```
  @Service
  @ServiceInstance(ServiceScope.SINGLETON) // equal to omitting the annotation
  class Fido implements Hound {}
  ```

152.10.7 Container Component Services

Beans, producer methods and producer fields that are `@ApplicationScoped` result in contextual instances that are shared throughout the CDI container. Therefore they can only provide singleton scoped services. Each such case results in a single service registration. The service object is the contextual instance created by the producer or bean. However, `@ApplicationScoped` beans can implement `org.osgi.framework.ServiceFactory` or `org.osgi.framework.PrototypeServiceFactory` in order to provide bundle or prototype scoped service objects.

Beans, producer methods and producer fields that are `@Dependent` result in contextual instances which are never shared in that a new contextual instance is created for each caller. Therefore they can provide services of all scopes as outlined in *Service Scope* on page 1220. The service object is a contextual instance created by the producer or bean on each request for a service object.

The use of `@ServiceInstance` on `@ApplicationScoped` beans will result in a definition error.

152.10.8 Single Component Services

Single components can only apply the `@Service` annotation to beans marked with `@SingleComponent`.

A single component providing a service results in a single service registration.

Service objects provided by the service registration are defined by the creation of contexts. In all cases, the service object provided is the contextual instance of the bean marked `@SingleComponent` obtained from the context.

152.10.9 Factory Component Services

Factory components can only apply the `@Service` annotation to beans marked with `@FactoryComponent`.

A factory component providing a service results in one service registration for every factory configuration object associated with the factory PID of the component.

Service objects provided by the service registration are defined by the creation of contexts. In all cases, the service object provided is the contextual instance of the bean marked `@FactoryComponent` obtained from the context.

152.11 Component Property Injection Points

A bean specifies injection of component properties using the @ComponentProperties annotation at an injection point.

The type typically associated with component properties is `java.util.Map<String, Object>`:

```
@Inject
@ComponentProperties
Map<String, Object> componentProperties;
```

However, component properties can be automatically converted to any type compatible with the conversions defined by the *Converter Specification* on page 1457.

Given the following configuration properties:

```
pool.name (String)
min.threads (int)
max.threads (int)
keep.alive.timeout (long)
```

The following example demonstrates conversion of component properties into a type safe object with defaults.

```
public static @interface PoolConfig {
 String pool_name();
 int min_threads() default 2;
 int max_threads() default 10;
 long keep_alive_timeout() default 500;
}

@Inject
@ComponentProperties
PoolConfig poolConfig;
```

Using @Reference in conjunction with `@ComponentProperties` will result in a definition error.

152.11.1 Coordinator Support

The *Coordinator Service Specification* on page 613 defines a mechanism for multiple parties to collaborate on a common task without *a priori* knowledge of who will collaborate in that task. Like *Configuration Admin Service Specification* on page 65, CCR must participate in such scenarios to coordinate with provisioning or configuration tasks.

If configuration changes occur and an implicit coordination exists, CCR must delay taking action on the configuration changes until the coordination terminates, regardless of whether the coordination fails or terminates regularly.

152.12 Reference Injection Points

Any injection point annotated with @Reference declares a service dependency.

152.12.1 Reference injection point types

Injection points specifying `@Reference` are limited to one of the following *injection point types* as representations of the dependent service(s). Given that type S is a type under which a service is published, the following *injection point types* are supported:

Table 152.6 Reference injection point types

Injection Point Type	Description

S	`// S = Dog`
	`@Inject`
	`@Reference`
	`Dog dog;`

org.osgi.framework.ServiceReference<S>	`// S = Dog`
	`@Inject`
	`@Reference`
	`ServiceReference<Dog> dog;`

java.util.Map<String, ? | Object>	In this case the `@Reference` annotation must specify service type S using it's value property.
	
	`// S = Dog`
	`@Inject`
	`@Reference(Dog.class)`
	`Map<String, Object> dogProperties;`
	
	Failure to specify the type in this scenario results in a definition error.

java.util.Map.Entry<Map<String, ? | Object>, S>	Represents a *tuple* containing the map of service properties as the key and the service instance as the value.
	
	`// S = Dog`
	`@Inject`
	`@Reference`
	`Map.Entry<Map<String, ?>, Dog> dog;`

BeanServiceObjects<S>	`// S = Dog`
	`@Inject`
	`@Reference`
	`BeanServiceObjects<Dog> dogs;`

S must be a concrete service type. The OSGi service registry does not support generics, therefore S cannot specify a generic type.

A definition error will result if any other types are used with injection points marked @Reference unless otherwise specified by this specification.

152.12.2 Reference Service scope

For a bound service, CCR must get the service object from the OSGi Framework's service registry using the `getService` method on the component's Bundle Context. If the service object for a bound service has been obtained and the service becomes unbound, CCR must unget the service object using the `ungetService` method on the component's Bundle Context and discard all references to the service object. This ensures that the bundle will only be exposed to a single instance of the service object at any given time.

For a bound service of a reference where the PrototypeRequired annotation was specified, only services registered with prototype service scope can be considered as target services. This ensures that each component instance can be exposed to a single, distinct instance of the service object. Using `@PrototypeRequired` effectively adds `service.scope=prototype` to the target property of the reference. A service that does not use prototype service scope cannot be used as a bound service for a reference with `@PrototypeRequired` since the service cannot provide a distinct service object for each component instance.

```
@Inject
@PrototypeRequired
@Reference
Hound hound;
```

152.12.3 Bean Service Objects

A Bean Service Objects for the bound service, can be used to obtain the actual service object or objects. This approach is useful when the referenced service has prototype service scope and the component instance needs multiple service objects for the service.

```
@Inject
@PrototypeRequired
@Reference
BeanServiceObjects<Hound> hounds;
```

The `@PrototypeRequired` annotation is optional. See *Service Scope* on page 1220.

152.12.4 Reference Greediness

References are greedy by default which means that higher ranking matches are immediately bound. Use the `@Reluctant` annotation to indicate that higher ranking matches should not bind once the reference has been resolved. Note that in the case of static references the component will be destroyed and recreated in order to immediately apply the better match. In the case of the container component, this will result in the entire CDI container being destroyed and recreated.

A static, greedy reference:

```
@Inject
@Reference
Hound hound;
```

A static, reluctant reference:

```
@Inject
@Reluctant
@Reference
Hound hound;
```

152.12.5 Service Type

As demonstrated earlier, it's possible to specify the service type of the reference by using @Reference.value() property. This supports use cases like java.util.Map<String, ?> where the service type cannot be determined.

```
@Inject
@Reference(Hound.class)
Map<String, Object> properties;
```

This makes it possible to target a more specific service type. A reference injection point whose type is Dog may target a service of type BassetHound:

```
@Inject
@Reference(BassetHound.class)
Dog dog;
```

The injection point type must be compatible with the service type. Otherwise a definition error will result.

152.12.6 Any Service Type

A special exception to the service type rules is defined when the special marker type Reference.Any is set as @Reference.value. This allows for any service to match the reference. However, the following criteria must be satisfied:

1. @Reference.value must specify the single value Reference.Any.class
2. @Reference.target must specify a valid, non-empty filter value
3. The injection point *service type* must be java.lang.Object. For example:

   ```
   @Inject
   @Reference(value = Reference.Any.class, target = "(foo=bar)")
   Optional<Object> match;
   ```

   or

   ```
   @Inject
   @Reference(value = Reference.Any.class, target = "(foo=bar)")
   List<Object> matches;
   ```

   Note that there may be performance impacts resulting from matching too broad a set of services. By definition the above list example with a target filter equal to (service.id=*) is perfectly valid but will match all services in the registry which will likely neither be very useful nor performant.

152.12.7 Target Filter

Target services for a reference are constrained by the reference's service type and the target property. A default target filter can be applied by specifying @Reference.target() property.

For example, a component wants to track all Dog services that have a service property service.vendor whose value is equal to Acme, Ltd.:

```
@Inject
@Reference(target = "(service.vendor=Acme, Ltd.)")
Collection<Dog> dogs;
```

152.12.7.1 Bean Property Types as target filters

Annotations meta-annotated with BeanPropertyType appearing on an injection point in conjunction with the @Reference annotation will further enhance the target filter as described by the rules for converting *Bean Property Types* on page 1213 to a map of properties assembled into a filter String according to the following steps:

1. any key=array pairs are flattened into many key=scalar pairs, one pair for each array value
2. format every key=scalar pair using the production

   ```
   pair       ::= '(' pairKey '=' pairScalar ')'
   pairKey    ::= < key >
   pairScalar ::= < scalar >
   ```

   If scalar must contain one of the characters reverse solidus ('\' \u005C), asterisk ('*' \u002A), parentheses open ('(' \u0028) or parentheses close (')' \u0029), then these characters must be preceded with the reverse solidus ('\' \u005C) character. Spaces are significant in scalar. Space characters are defined by Character.isWhiteSpace()
3. concatenate all results of step 2. into a single String
4. append the value of @Reference.target() to the result of step 3.
5. format the result of step 4. using the production

   ```
   target ::= '(' '&' step4 ')'
   step4  ::= < result of step 4. >
   ```

Given the following example:

```
enum Tricks {
 SIT, STAND, SHAKE_PAW, TREAT_ON_NOSE
}
@Repeatable(...)
@BeanPropertyType
@interface Trick {
 Tricks value();
}

@Inject
@Reference(target = "(service.vendor=Acme Kennels, Ltd.)")
@Trick(SIT)
@Trick(TREAT_ON_NOSE)
Dog dog;
```

The target filter will be:

```
(&(trick=sit)(trick=treat_on_nose)(service.vendor=Acme Kennels, Ltd.))
```

152.12.8 Reference Names

The @javax.inject.Named annotation may be used to specify a name to serve as the base of the component properties used to configure the reference. If not specified the name of the reference will be derived from the fully qualified class name of the class defining the reference injection point and the reference injection point.

The production for generated names is:

```
name   ::= prefix '.' suffix
```

```
prefix ::= named | qname
named  ::= < @Named.value >
suffix ::= field | ctor | method
field  ::= < name of field >
ctor   ::= 'new' pIndex
method ::= mName pIndex
mName  ::= < method name >
pIndex ::= < index of @Reference parameter >
```

It is a definition error to have two references with the same name.

It is a definition error to specify the @javax.inject.Named annotation with no value.

In the following example the reference name is example.Fido.mate and the target and minimum cardinality properties of the reference will be example.Fido.mate.target and example.Fido.mate.cardinality.minimum respectively:

```
package example;

@SingleComponent
class Fido {
 @Inject
 @Reference
 Dog mate;
}
```

In the following example the reference name is foo and the target and minimum cardinality properties of the reference will be foo.target and foo.cardinality.minimum respectively:

```
package example;

@SingleComponent
class Fido {
 @Inject
 @Named("foo")
 @Reference
 Dog mate;
}
```

152.12.9 Static References

Static references are the most common form of reference injection point. Static means that their values do not change during the lifetime of the component instance which means that in order to change the service bound to the reference injection point, the entire component instance must be destroyed and recreated.

The following are more examples of static reference injection points:

```
@Inject
@Reference
Dog dog;

@Inject
@Reference(BassetHound.class)
Map<String, Object> props;

@Inject
void setHounds(@Reference BeanServiceObjects<Hound> hounds) {...}
```

```
@Inject
@Reference
ServiceReference<Spot> spot;
```

Static reference injection points are *mandatory* by default. They require a number of services equal to or greater than their minimum cardinality to be available in order for the component instance to resolve.

152.12.10 Static Optional References

Optional reference injection points allow a component instance to become resolved when fewer matching services are found than required by the reference's minimum cardinality. The injection point type must be java.util.Optional<R> where R is one of the supported reference injection point types.

The following are examples of static optional references:

```
@Inject
@Reference
Optional<Dog> dog;

@Inject
@Reference(BassetHound.class)
Optional<Map<String, Object>> props;

@Inject
void setHounds(@Reference Optional<BeanServiceObjects<Hound>> hounds) {...}

@Inject
@Reference
Optional<ServiceReference<Spot>> spot;
```

As with other static references, static means that their values do not change during the lifetime of the component instance which means that in order to change the service bound to the reference injection point, the entire component instance must be destroyed and recreated.

152.12.11 Static Multi-cardinality References

Multi-cardinality references are specified using an injection point type of java.util.Collection<R>, or java.util.List<R> where R is one of the supported reference injection point types. Repeating the static examples as multi-cardinality references, we get:

```
@Inject
@Reference
List<Dog> dogs;

@Inject
@Reference(BassetHound.class)
Collection<Map<String, Object>> props;

@Inject
void setHounds(@Reference List<BeanServiceObjects<Hound>> hounds) {...}

@Inject
@Reference
Collection<ServiceReference<Spot>> spots;
```

Multi-cardinality references are naturally *optional* since the default value of the minimum cardinality property is 0. See *Minimum Cardinality Property* on page 1212.

As with other static references, static means that their values do not change during the lifetime of the component instance which means that in order to change the services bound to the reference injection point, the entire component instance must be destroyed and recreated.

152.12.12 Default Minimum Cardinality

As stated in *Minimum Cardinality Property* on page 1212 every reference has a configurable reference property name.cardinality.minimum. However, there are cases where it is appropriate to specify a non-zero default minimum cardinality. The MinimumCardinality annotation provides this functionality.

The following is an example of setting the minimum cardinality:

```
@Inject
@MinimumCardinality(3)
@Reference
List<Dog> guards;
```

The value must be a positive integer.

Specifying this annotation on a unary reference results in a definition error.

152.12.13 Dynamic References

Dynamic reference injection points are specified using an injection point type of javax.inject.Provider<R> where R is one of the supported reference injection point types, java.util.Optional<R>, java.util.Collection<R>, or java.util.List<R>.

The following are examples of dynamic references:

```
@Inject
@Reference
Provider<Dog> dog;

@Inject
@Reference(BassetHound.class)
Provider<Collection<Map<String, Object>>> props;

@Inject
void setHounds(
  @Reference
  Provider<List<BeanServiceObjects<Hound>>> hounds
 ) {...}

@Inject
@Reference
Provider<Optional<ServiceReference<Spot>>> spots;
```

The evaluation of javax.inject.Provider.get() is performed such that each invocation may produce a different result except for returning null.

Specifying the @MinimumCardinality annotation with a non-zero value on a dynamic, multi-cardinality reference results in the component not being resolved until the number of matching services becomes equal to or greater than the specified minimum cardinality.

152.13 Interacting with Service Events

It is often necessary to observe the addition, modification and removal of services from the service registry. This specification provides 3 special bean types, referred to as *binder types*, which make it possible to bind methods to coordinate across the service events of set of services. The type argument S indicates the service type expected unless further reduced as described by *Service Type* on page 1224. Bean Property Types may also be used to expand the target filter as defined in *Bean Property Types as target filters* on page 1225.

- BindService<S> - The `BindService` bean allows for coordination of service events when the service instance is required.
- BindBeanServiceObjects<S> - The `BindBeanServiceObjects` bean allows for coordination of service events when bean service objects are required.
- BindServiceReference<S> - The `BindServiceReference` bean allows for coordination of service events when the service reference is required.

These bean types declare a builder style interface for binding the necessary methods to coordinate the events. The following example binds service event methods over the set of services whose type is `Dog` and having the service property `service.vendor=Acme Inc.`:

```
@Inject
@ServiceVendor("Acme Inc.")
void bindDogs(BindService<Dog> binder) {
 binder.
  adding(this::adding).
  modified(this::modified).
  removed(this::removed).
  bind();
}

void adding(Dog dog, Map<String,Object> properties) {...}
void modified(Dog dog, Map<String,Object> properties) {...}
void removed(Dog dog, Map<String,Object> properties) {...}
```

The terminal `bind()` method must be called to inform CCR that the bind process is complete. Binding a subset of methods is allowed. Only the last bind method specified for any given service event will be used. For example, given the following invocation:

```
@Inject
void bindDogs(BindService<Dog> binder) {
 binder.
  adding(this::addingA).
  adding(this::addingB).
  bind();
}
```

only the method `addingB` will be used.

An example of a binder type injected into a field:

```
@Inject
void bindDogs(BindBeanServiceObjects<Dog> binder) {
 binder.
  adding(this::adding).
  removed(this::removed).
  bind();
}

void adding(BeanServiceObjects<Dog> dogs) {...}
void removed(BeanServiceObjects<Dog> dogs) {...}
```

Binder objects are @Dependent objects and are not thread safe. They are intended to be used during the creation phase of component beans before the end of the [10] *@PostConstruct* method. Executing any binder object method after this time will result in unspecified behavior.

152.14 CDI Component Runtime

CDI Component Runtime (CCR) is the actor that manages the CDI containers and their life cycle and allows for their introspection.

152.14.1 Relationship to the OSGi Framework

CCR must have access to the Bundle Context of any CDI bundle. CCR needs access to the Bundle Context for the following reasons:

- To be able to register and get services on behalf of a CDI bundle.
- To interact with the Configuration Admin on behalf of a CDI bundle.
- To interact with the Log Service on behalf of a CDI bundle.
- To make the Bundle Context available for injection in the CDI bundle's beans.

CCR should use the Bundle.getBundleContext() method to obtain the Bundle Context reference.

152.14.2 Injecting the Bundle Context

The Bundle Context of the CDI bundle can be injected. The injection point must be of type org.osgi.framework.BundleContext and must not specify any qualifiers.

```
@Inject
BundleContext bundleContext;
```

152.14.3 Starting and Stopping CCR

When CCR is implemented as a bundle, any containers activated by CCR must be deactivated when the CCR bundle is stopped. When the CCR bundle is started, it must process the CDI metadata declared in CDI bundles. This includes bundles which are started and are awaiting lazy activation.

152.14.4 Logging Messages

When CCR must log a message to the Log Service, it must use a Logger named using the component's name and associated with the CDI bundle. To obtain the Logger object, CCR must call the LoggerFactory.getLogger(Bundle bundle, String name, Class loggerType) method passing the CDI bundle as the first argument and the name of the component as the second argument. If CCR cannot know the component name, because the error is not associated with a component or the error occurred before the component template is processed, then CCR must use the bundle's Root Logger, that is, the Logger named ROOT.

152.14.5 Bundle Activator Interaction

A CDI bundle may also declare a Bundle Activator. Such a bundle may also be marked for lazy activation. Since CDI containers are activated by CCR and Bundle Activators are called by the OSGi

Framework, a bundle using both a CDI container and a Bundle Activator must take care. The Bundle Activator's start method must not rely upon CCR having activated the bundle's CDI container. However, the CDI container can rely upon the Bundle Activator's start method having been called. That is, there is a *happens-before* relationship between the Bundle Activator's start method being run and the CDI container being activated.

152.14.6 Introspection

CCR provides an introspection API for examining the runtime state of the CDI bundles processed by CCR. CCR must register a CDIComponentRuntime service upon startup. The CDI Component Runtime service provides methods to inspect CDI containers. The service uses *Data Transfer Objects (DTO)* as arguments and return values. The rules for Data Transfer Objects are specified in *OSGi Core Release 8* on page 19.

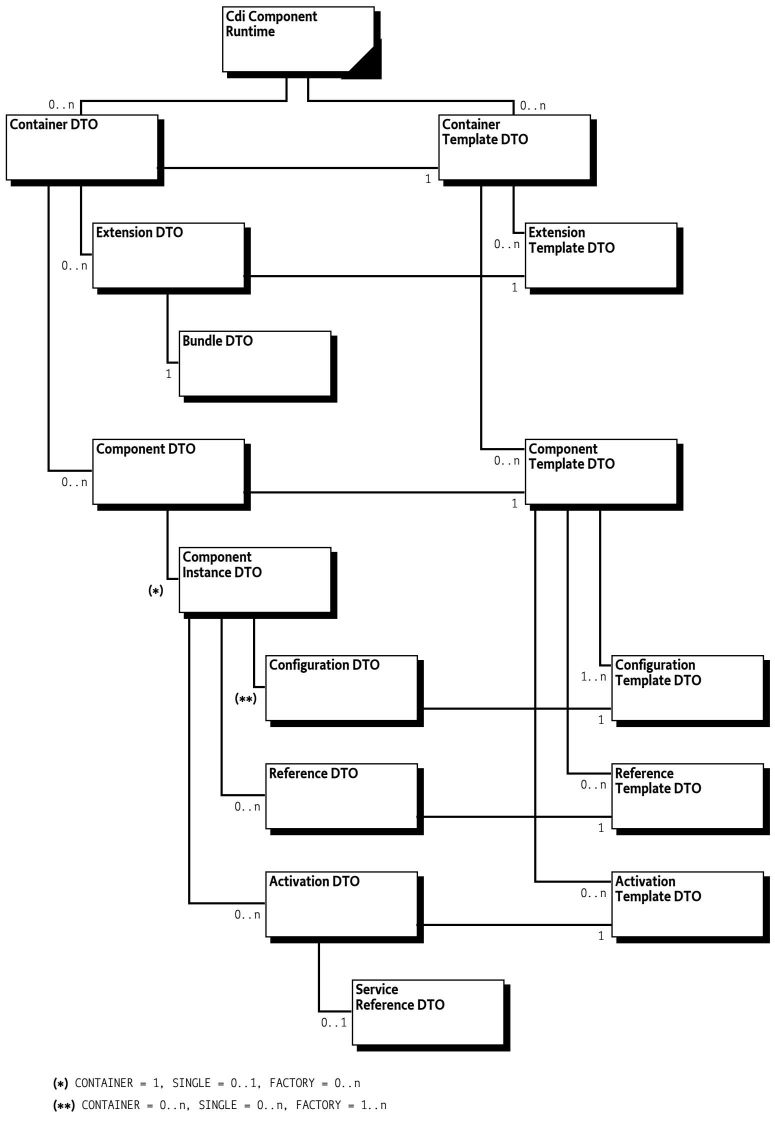
The CDI Component Runtime service provides the following methods.

- getContainerDTOs(Bundle...) - For each specified bundle, if the bundle is active and processed by CCR, and the bundle is a valid CDI bundle, the returned collection will contain a ContainerDTO describing the CDI container.
- getContainerTemplateDTO(Bundle) - If the specified bundle is active and processed by CCR, and the bundle is a valid CDI bundle, the method will return a ContainerTemplateDTO describing the template metadata of the CDI container.

The runtime state of the containers can change at any time. So any information returned by these methods only provides a snapshot of the state at the time of the method call.

There are a number of DTOs available via the CDI Component Runtime service.

Figure 152.3 *CDI Component Runtime DTOs*



The ContainerDTO specifies a changeCount field of type long. Whenever the DTOs bellow the ContainerDTO change, CCR will increment the ContainerDTO's changeCount. Whenever any ContainerDTO changes, CCR will update the service.changecount service property of the CDIComponentRuntime service. CCR may use a single update to the service.changecount property to reflect updates in multiple ContainerDTOs. See org.osgi.framework.Constants.SERVICE_CHANGECOUNT in *OSGi Core Release 8*.

152.14.7 Logger Support

CCR provides special support for logging via the Log Service specification. CCR must provide @Dependent objects of type org.osgi.service.log.Logger and org.osgi.service.log.FormatterLogger.

To obtain the Logger object for injection, CCR must call the LoggerFactory.getLogger(Bundle bundle, String name, Class loggerType) method passing the bundle declaring the component as the first argument, the fully qualified name of the injection point's declaring class as the second argument, and the type of the injection point; org.osgi.service.log.Logger or org.osgi.service.log.FormatterLogger, as the third argument. The typical usage is:

```
@Inject
Logger logger;

@PostConstruct
void init() {
 logger.debug("Initialized");
}
```

Another example using method injection along with component properties (coerced to Config):

```
public static @interface Config {
 String component_name();
}

@Inject
void setup(@ComponentProperties Config config, Logger logger) {
 logger.trace("Activating component {}", config.component_name());
}
```

152.14.8 Disabling Components

All components in a CDI bundle are enabled by default. However, any component can be disabled through configuration using the single configuration object associated with the container PID by defining a property using the component name suffixed with .enabled. The value's type is boolean.

```
enabled  ::= compName '.enabled'
compName ::= < component name >
```

The following is an example disabling a component whose name is foo:

```
foo.enabled=false
```

The container component can be disabled using it's component name, which is the *container id*. As a result of disabling the container component, all components in the CDI bundle are also disabled.

152.14.9 Container Component and Service Cycles

There is no special support to allow service cycles within the *container component*. CDI provides existing mechanisms for wiring and collaborating within the CDI container. However, if an container component defines a dynamic, optional reference, then a service subsequently provided by the container component may satisfy the reference at some point when the container component is satisfied. However, if the reference is static and mandatory and the only potentially matching service is one provided by the container component itself, then the container component would wait forever for a service that will never arrive. This is simple design error. The information about unsatisfied references is available from the CDIComponentRuntime service.

152.15 Capabilities

CCR must provide the following capabilities.

A capability in the osgi.extender namespace declaring an extender with the name osgi.cdi. In addition to the specification packages, this capability must declare a uses constraint for the javax.inject package. For example:

```
Provide-Capability:
```

```
osgi.extender;
  osgi.extender="osgi.cdi";
  version:Version="1.0";
  uses:="javax.inject, org.osgi.service.cdi, org.osgi.service.cdi.annotations,
    org.osgi.service.cdi.reference, org.osgi.service.cdi.runtime,
    org.osgi.service.cdi.runtime.dto,
    org.osgi.service.cdi.runtime.dto.template"
```

This capability must follow the rules defined for the *osgi.extender Namespace* on page 707.

A CDI bundle must require the osgi.extender capability from CCR. This requirement will wire the bundle to the CCR implementation and ensure that CCR is using the same org.osgi.service.cdi.* packages as the bundle if the bundle uses those packages.

```
Require-Capability:
 osgi.extender;
  filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"
```

CCR must only process a CDI bundle if the bundle's wiring has a required wire for at least one osgi.extender capability with the name osgi.cdi and the first of these required wires is wired to CCR.

When using the annotations Bean or Beans, the above requirement is automatically added to the manifest when the code is processed by a supporting build tool capable of interpreting *Bundle Annotations* defined in *OSGi Core Release 8* on page 19.

The requirement may be specified directly on any class or package in the CDI bundle by using the RequireCDIExtender annotation when the code is processed by a supporting build tool capable of interpreting *Bundle Annotations* defined in *OSGi Core Release 8* on page 19.

Specifying CDI bean descriptors - As specified in *Bean Descriptors* on page 1235 a CDI bundle must declare all CDI bean descriptors CCR is expected to operate on. This is done by adding the attribute descriptor, of type List<String>, to the requirement.

Specifying the list of bean classes - As specified in *Bean Discovery* on page 1235 a CDI bundle must declare all bean classes CCR is expected to operate on. This is done by adding the attribute beans, of type List<String>, to the requirement.

A capability in the osgi.implementation namespace declaring an implementation with the name osgi.cdi. In addition to the specification packages, this capability must also declare a uses constraint for the javax.enterprise.* packages. For example:

```
Provide-Capability:
 osgi.implementation;
  osgi.implementation="osgi.cdi";
  version:Version="1.0";
  uses:="javax.enterprise.context, javax.enterprise.context.control,
   javax.enterprise.context.spi, javax.enterprise.event,
   javax.enterprise.inject, javax.enterprise.inject.literal,
   javax.enterprise.inject.spi, javax.enterprise.inject.spi.configurator,
   javax.enterprise.util, org.osgi.service.cdi,
   org.osgi.service.cdi.annotations,
   org.osgi.service.cdi.reference, org.osgi.service.cdi.runtime,
   org.osgi.service.cdi.runtime.dto,
   org.osgi.service.cdi.runtime.dto.template"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

A capability in the osgi.service namespace representing the CDIComponentRuntime service. This capability must also declare a uses constraint for the org.osgi.service.cdi.runtime package. For example:

```
Provide-Capability:
 osgi.service;
  objectClass:List<String>=
   "org.osgi.service.cdi.runtime.CDIComponentRuntime";
  uses:="org.osgi.service.cdi.runtime"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

A capability in the osgi.service namespace for every service declared by the metadata in the CDI bundle.

152.16 Relationship to CDI features

CDI has many features which may occasionally interact with the OSGi CDI integrations defined by this specification.

152.16.1 Bean Descriptors

The [6] *Packaging and deployment* chapter of the CDI specification defines XML descriptors which are used to control the CDI container. This specification expects that these descriptors be declared using the osgi.cdi extender requirement attribute descriptor of type List<String>. For example:

```
Require-Capability:
 osgi.extender;
  filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0.0)))";
  descriptor:List<String>="META-INF/beans.xml"
```

If the attribute is not specified the default value of META-INF/beans.xml is used.

CCR must find descriptors by calling Bundle.getResources(String) for each specified value. Note that the accepted syntax for the values is the same as for java.lang.ClassLoader.getResources. See osgi.cdi extender capability.

152.16.2 Bean Discovery

The CDI specification defines 3 bean discover modes which perform runtime class discovery:

- *all* - All classes in the jar are passed to the CDI container and processed.
- *none* - No classes in the jar are passed to the CDI container. It is assumed however that *portable extensions* may yet provide beans.
- *annotated* (default) - Only classes matching the definition of *annotated beans* as defined by the [4] *Default bean discovery mode* are passed to the CDI container and processed.

This specification avoids runtime class analysis concern by ignoring the bean discovery mode specified or implied by the descriptors, requiring bean classes to be pre-calculated at build time such that the CDI container receives a concrete list of classes to process.

It is expected that the aforementioned bean discover modes be implemented in build tooling and be performed at build time.

A CDI bundle must specify the list of classes to process using the osgi.cdi extender requirement attribute beans of type List<String>. For example:

```
Require-Capability:
 osgi.extender;
  filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0.0)))";
  beans:List<String>="org.foo.Bar, org.foo.baz.Fum"
```

See osgi.cdi extender capability.

152.16.2.1 Build tool support

The bean descriptors specified by the CDI specification allow for narrowing the range of processed classes by defining [5] *Exclude filters*. While these filters are still considered, they are only applied over the concrete list of classes passed from the beans attribute.

Build tools may opt to implement bean discover modes. Implementing the discovery mode *all* simply requires placing the names of all classes found in the bundle in the beans attribute. Implementing the discovery mode *annotated* involves collecting the names of all classes matching the definition of *annotated beans* as defined by the [4] *Default bean discovery mode* and placing those in the beans attribute.

Another option is to use the CLASS retention annotation defined by this specification.

The CLASS retention annotation Bean may be applied to a class to indicate to supporting build tools it must be included in the beans list.

The CLASS retention annotation Beans may be applied to a package to indicate to supporting build tools that all classes in the package must be included in the beans list.

Specifying a value indicates to supporting build tools that the specified classes in the package must be included in the beans list.

152.16.3 Portable Extensions

CDI Portable Extensions use CDI's SPI which provides a powerful mechanism for extending the base functionality of CDI. Portable extensions may add, modify or read bean and bean class metadata, define custom contexts, and much more. Through the SPI a portable extension can participate in all aspects of the CDI Container's life cycle.

Portable extensions must be provided as OSGi services using the interface javax.enterprise.inject.spi.Extension. Portable extension services must specify the service property osgi.cdi.extension whose value is a name identifying the functionality provided by the portable extension.

Table 152.7 Portable Extension Service Properties

Service Property	Type	Description
osgi.cdi.extension	String	The name of the Portable Extension

For example, a portable extension service that provides an implementation of the [14] *Java Transaction API* should specify the value of it's osgi.cdi.extension service property using the [15] *Portable Java Contract* name specified for it, which is JavaJTA.

Portable Extension bundles must define a capability using the namespace osgi.cdi.extension having an attribute osgi.cdi.extension whose value is the same as the name specified in the osgi.cdi.extension service property of the portable extension service. The capability must also specify a version attribute of type Version. The capability must also specify a uses directive listing all of the Java packages provided as part of the Portable Extension's API. If the portable extension implements an API specified as a Portable Java Contract the uses list should match the Portable Java Contract.

```
Provide-Capability:
 osgi.cdi.extension;
  osgi.cdi.extension=JavaJTA;
  version:Version="1.2";
  uses:="javax.transaction,javax.transaction.xa"
```

CDI bundles express a dependency on a portable extension by specifying a requirement in the osgi.cdi.extension namespace whose filter matches a portable extension capability. For example:

```
Require-Capability:
 osgi.cdi.extension;
  filter:="(&(osgi.cdi.extension=JavaJTA)(version=1.2))";
```

See osgi.cdi extender capability.

A Portable extension bundle must require the osgi.implementation capability from CCR. This requirement will wire the extension bundle to the CCR implementation and ensure that CCR is using the same javax.enterprise.* packages as the portable extension bundle.

```
Require-Capability:
 osgi.implementation;
  filter:="(&(osgi.implementation=osgi.cdi)(version>=1.0)(!(version>=2.0)))"
```

The requirement may be specified directly on any class in the portable extension bundle by using the RequireCDIImplementation annotation when the code is processed by tooling capable of interpreting *Bundle Annotations* defined in *OSGi Core Release 8* on page 19.

152.16.3.1 Portable Extension Services and Beans

Portable extension bundles intending to provide additional beans must do so programmatically using the SPI. Bean descriptors in the bundle providing the portable extension service are not visible to the CDI container and therefore play no role in bean discovery.

152.16.3.2 Embedded Portable Extension

Portable extensions which are embedded in the CDI bundle are discoverable through the CDI specified *service loader* mechanism using the class loader of the CDI bundle.

152.16.4 Bean Manager

When the container component is satisfied CCR must published the CDI container's javax.enterprise.inject.spi.BeanManager to the service registry using the ServiceContext of the CDI bundle accompanied by the following service property:

Table 152.8 *Bean Manager Service Properties*

Service Property	Type	Description
osgi.cdi.container.id	String	The container id. The constant CDI_CONTAINER_ID_PROPERTY exists for convenience. See *Container Component* on page 1206.

The javax.enterprise.inject.spi.BeanManager must be unregistered when the container component becomes unsatisfied.

152.16.5 Decorators and Interceptors

Decorators and Interceptors are used to wrap contextual instances with proxies to deliver additional, targeted functionality. However, these features do not support [3] *unproxyable bean types*. Attempting to apply either feature to a bean or producer having an unproxyable bean type will result in a definition error. This limitation extends to CCR where applicable. The @javax.enterprise.inject.Typed annotation is available to explicitly reduce the set of bean types, making it possible to use either feature on beans having unproxyable types. Implementations of this specification must support the use of @javax.enterprise.inject.Typed when publishing services.

Service objects are the product of beans and producers. As such they may be targeted by Decorators and/or Interceptors and wrapped by proxies. Therefore the subset of types under which the service is published must be a subset of the bean types, including further restrictions declared by @javax.enterprise.inject.Typed. Service types not contained in the restricted set of bean types will result in a definition error. See *@Service Type Restrictions* on page 1219.

152.17 Security

When Java permissions are enabled, CCR must perform the following security procedures.

152.17.1 Service Permissions

CCR dependencies are built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish, find or bind services.

If a component specifies a service, that component cannot be satisfied unless the CDI bundle has ServicePermission[<provides>, REGISTER] for each provided interface specified for the service.

If a component's reference does not specify optional cardinality, the reference cannot be satisfied unless the CDI bundle has ServicePermission[<interface>, GET] for the specified interface in the reference. If the reference specifies optional cardinality but the component's bundle does not have ServicePermission[<interface>, GET] for the specified interface in the reference, no service must be bound for this reference.

CCR must have ServicePermission[CDIComponentRuntime, REGISTER] permission to register the CDIComponentRuntime service. Administrative bundles wishing to use the CDIComponentRuntime service must have ServicePermission[CDIComponentRuntime, GET] permission. In general, this permission should only be granted to administrative bundles to limit access to the potentially intrusive methods provided by this service.

152.17.2 Required Admin Permission

CCR requires AdminPermission[*,CONTEXT] because it needs access to the CDI bundle's Bundle Context object with the Bundle.getBundleContext() method.

152.17.3 Using hasPermission

CCR does all publishing, finding and binding of services on behalf of the component using the Bundle Context of the CDI bundle. This means that normal stack-based permission checks will check CCR and not the component's bundle. Since CCR is registering and getting services on behalf of a CDI bundle, CCR must call the Bundle.hasPermission method to validate that a CDI bundle has the necessary permission to register or get a service.

152.17.4 Configuration Multi-Locations and Regions

CCR must ensure a bundle has the proper ConfigurationPermission for a Configuration used by its components when the Configuration has a multi-location. See *Using Multi-Locations* on page 83 for more information on multi-locations and *Regions* on page 84 for more information on regions. If a bundle does not have the necessary permission for a multi-location Configuration, then CCR must act as if the Configuration does not exist for the bundle.

152.18 org.osgi.service.cdi

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.cdi; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.cdi; version="[1.0,1.1)"
```

152.18.1 Summary

- `CDIConstants` - Defines CDI constants.
- `ComponentType` - Define the possible values for ComponentTemplateDTO.type.
- `ConfigurationPolicy` - Defines the possible values for configuration policy.
- `MaximumCardinality` - Defines the possible values for maximum cardinality of dependencies.
- `ReferencePolicy` - Defines the possible values of the policy of a reference towards propagating service changes to the CDI runtime
- `ReferencePolicyOption` - Defines the possible values of the policy of a satisfied reference towards new matching services appearing.
- `ServiceScope` - Possible values for ActivationTemplateDTO.scope.

152.18.2 public class CDIConstants

Defines CDI constants.

Provider Type Consumers of this API must not implement this type

152.18.2.1 public static final String CDI_CAPABILITY_NAME = "osgi.cdi"

Capability name for CDI Integration.

Used in `Provide-Capability` and `Require-Capability` manifest headers with the `osgi.extender` namespace. For example:

```
Require-Capability: osgi.extender; «
  filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"
```

152.18.2.2 public static final String CDI_COMPONENT_NAME = "$"

Special string representing the name of a Component.

This string can be used with PID OR factory PID to specify the name of the component.

For example:

```
@PID(CDI_COMPONENT_NAME)
```

152.18.2.3 public static final String CDI_CONTAINER_ID = "container.id"

The attribute of the CDI extender requirement declaring the container's id.

```
Require-Capability: osgi.extender; «
  filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"; «
  container.id="my.container"
```

152.18.2.4 public static final String CDI_CONTAINER_ID_PROPERTY = "osgi.cdi.container.id"

The key used for the container id service property in services provided by CCR.

152.18.2.5 public static final String CDI_EXTENSION_PROPERTY = "osgi.cdi.extension"

A service property applied to javax.enterprise.inject.spi.Extension services, whose value is the name of the extension.

152.18.2.6 public static final String CDI_SPECIFICATION_VERSION = "1.0"

Compile time constant for the Specification Version of CDI Integration.

Used in Version and Requirement annotations. The value of this compile time constant will change when the specification version of CDI Integration is updated.

152.18.2.7 public static final String REQUIREMENT_BEANS_ATTRIBUTE = "beans"

The 'beans' attribute on the CDI extender requirement.

The value of this attribute is a list of bean class names that will be processed by CCR. The default value is an empty list. For example:

```
Require-Capability: osgi.extender; «
 filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"; «
 beans:List<String>="com.acme.Foo,com.acme.bar.Baz"
```

152.18.2.8 public static final String REQUIREMENT_DESCRIPTOR_ATTRIBUTE = "descriptor"

The 'descriptor' attribute on the CDI extender requirement.

The value of this attribute is a list of bean CDI bean descriptor file paths to be searched on the Bundle-ClassPath. For example:

```
Require-Capability: osgi.extender; «
 filter:="(&(osgi.extender=osgi.cdi)(version>=1.0)(!(version>=2.0)))"; «
 descriptor:List<String>="META-INF/beans.xml"
```

152.18.3 enum ComponentType

Define the possible values for ComponentTemplateDTO.type.

152.18.3.1 CONTAINER

The component is the *Container Component*.

152.18.3.2 SINGLE

The component is an *Single Component*.

152.18.3.3 FACTORY

The component is an *Factory Component*.

152.18.3.4 public static ComponentType valueOf(String name)

152.18.3.5 public static ComponentType[] values()

152.18.4 enum ConfigurationPolicy

Defines the possible values for configuration policy.

152.18.4.1 OPTIONAL

Defines the optional configuration policy.

152.18.4.2 REQUIRED

Defines the required configuration policy.

152.18.4.3 **public static ConfigurationPolicy valueOf(String name)**

152.18.4.4 **public static ConfigurationPolicy[] values()**

152.18.5 enum MaximumCardinality

Defines the possible values for maximum cardinality of dependencies.

152.18.5.1 **ONE**

Defines a unary reference.

152.18.5.2 **MANY**

Defines a plural reference.

152.18.5.3 **public static MaximumCardinality fromInt(int value)**

value The integer representation of an upper cardinality boundary

□ Resolve an integer to an upper cardinality boundary.

Returns The enum representation of the upper cardinality boundary described by `value`

152.18.5.4 **public int toInt()**

□ Convert this upper cardinality boundary to an integer

Returns The integer representation of this upper cardinality boundary

152.18.5.5 **public static MaximumCardinality valueOf(String name)**

152.18.5.6 **public static MaximumCardinality[] values()**

152.18.6 enum ReferencePolicy

Defines the possible values of the policy of a reference towards propagating service changes to the CDI runtime

152.18.6.1 **STATIC**

Reboot the CDI component that depends on this reference

152.18.6.2 **DYNAMIC**

Update the CDI reference

152.18.6.3 **public static ReferencePolicy valueOf(String name)**

152.18.6.4 **public static ReferencePolicy[] values()**

152.18.7 enum ReferencePolicyOption

Defines the possible values of the policy of a satisfied reference towards new matching services appearing.

152.18.7.1 **GREEDY**

Consume the matching service applying it's ReferencePolicy

152.18.7.2 **RELUCTANT**

Do not consume the matching service

152.18.7.3 public static ReferencePolicyOption valueOf(String name)

152.18.7.4 public static ReferencePolicyOption[] values()

152.18.8 enum ServiceScope

Possible values for ActivationTemplateDTO.scope.

152.18.8.1 SINGLETON

This activation will only ever create one instance

The instance is created after the parent component becomes satisfied and is destroyed before the parent component becomes unsatisfied.

If ActivationTemplateDTO.serviceClasses is not empty the instance will be registered as an OSGi service with `service.scope=singleton`.

152.18.8.2 BUNDLE

This activation will register an OSGi service with `service.scope=bundle`.

The service is registered just after all SINGLETON activations are set up and just before all SINGLETON activations are torn down.

The ActivationTemplateDTO.serviceClasses is not empty when this scope is used.

152.18.8.3 PROTOTYPE

This activation will register an OSGi service with `service.scope=prototype`.

The service is registered just after all SINGLETON activations are set up and just before all SINGLETON activations are torn down.

The ActivationTemplateDTO.serviceClasses is not empty when this scope is used.

152.18.8.4 public static ServiceScope valueOf(String name)

152.18.8.5 public static ServiceScope[] values()

152.19 org.osgi.service.cdi.annotations

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.cdi.annotations; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.cdi.annotations; version="[1.0,1.1)"`

152.19.1 Summary

- `Bean` - Annotation used to indicate that build tooling must be included the class in the `osgi.cdi` beans list.

- BeanPropertyType - Identify the annotated annotation as a Bean Property Type.
- BeanPropertyType.Literal - Support inline instantiation of the BeanPropertyType annotation.
- Beans - Annotation used to indicate that build tooling must be included the specified classes in the osgi.cdi beans list.
- ComponentProperties - Annotation used with Inject in order to have component properties injected.
- ComponentProperties.Literal - Support inline instantiation of the ComponentProperties annotation.
- ComponentScoped - This scope is used to declare a bean who's lifecycle is determined by the state of it's OSGi dependencies and the SingleComponent(s) and FactoryComponent(s) that may reference it through injection.
- ComponentScoped.Literal - Support inline instantiation of the ComponentScoped annotation.
- FactoryComponent - Identifies a factory component.
- FactoryComponent.Literal - Support inline instantiation of the FactoryComponent annotation.
- MinimumCardinality - Annotation used in conjunction with Reference to specify the minimum cardinality reference property.
- MinimumCardinality.Literal - Support inline instantiation of the MinimumCardinality annotation.
- PID - Annotation used in collaboration with ComponentScoped to specify singleton configurations and their policy.
- PID.Literal - Support inline instantiation of the PID annotation.
- PIDs - Annotation used in conjunction with ComponentScoped in order to associate configurations with the component bean.
- PIDs.Literal - Support inline instantiation of the PIDs annotation.
- PrototypeRequired - Used with @Reference, BindService, BindBeanServiceObjects and BindServiceReference to indicate that the service must be service.scope=prototype.
- PrototypeRequired.Literal - Support inline instantiation of the PrototypeRequired annotation.
- Reference - Annotation used on injection points informing the CDI container that the injection should apply a service obtained from the OSGi registry.
- Reference.Any - A marker type used in Reference.value to indicate that a reference injection point may accept any service type(s).
- Reference.Literal - Support inline instantiation of the Reference annotation.
- Reluctant - Annotation used to indicate that the behavior of the reference should be reluctant.
- Reluctant.Literal - Support inline instantiation of the Reluctant annotation.
- RequireCDIExtender - This annotation can be used to require the CDI Component Runtime extender.
- RequireCDIImplementation - This annotation can be used to require the CDI Component Runtime implementation.
- Service - Annotation used to specify that a bean should be published as a service.
- Service.Literal - Support inline instantiation of the Service annotation.
- ServiceInstance - Annotation used on beans, observer methods and observer fields to specify the service scope for the service.
- ServiceInstance.Literal - Support inline instantiation of the ServiceInstance annotation.
- SingleComponent - Identifies a single component.
- SingleComponent.Literal - Support inline instantiation of the SingleComponent annotation.

152.19.2 @Bean

Annotation used to indicate that build tooling must be included the class in the osgi.cdi beans list.

Retention CLASS

Target TYPE

152.19.3 @BeanPropertyType

Identify the annotated annotation as a Bean Property Type.

Bean Property Type can be applied to beans annotated with SingleComponent, FactoryComponent, to beans annotated with ApplicationScoped or Dependent where the Service annotation is applied, to methods and fields marked as Produces where the Service annotation is applied, or to injection points where the Reference annotation is applied.

See Also Bean Property Types.

Retention RUNTIME

Target ANNOTATION_TYPE

152.19.4 public static final class BeanPropertyType.Literal extends AnnotationLiteral<BeanPropertyType> implements BeanPropertyType

Support inline instantiation of the BeanPropertyType annotation.

152.19.4.1 public static final BeanPropertyType INSTANCE

Default instance.

152.19.4.2 public Literal()

152.19.5 @Beans

Annotation used to indicate that build tooling must be included the specified classes in the osgi.cdi beans list.

Retention CLASS

Target PACKAGE

152.19.5.1 Class<?>[] value default {}

□ Specify the list of classes from the current package. Specifying no value (or an empty array) indicates to include all classes in the package.

152.19.6 @ComponentProperties

Annotation used with Inject in order to have component properties injected.

See "Component Properties".

Retention RUNTIME

Target FIELD, PARAMETER

152.19.7 public static final class ComponentProperties.Literal extends AnnotationLiteral<ComponentProperties> implements ComponentProperties

Support inline instantiation of the ComponentProperties annotation.

152.19.7.1 public static final ComponentProperties INSTANCE

Default instance.

152.19.7.2 public Literal()

152.19.8 @ComponentScoped

This scope is used to declare a bean who's lifecycle is determined by the state of it's OSGi dependencies and the SingleComponent(s) and FactoryComponent(s) that may reference it through injection.

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.19.9 public static final class ComponentScoped.Literal extends AnnotationLiteral<ComponentScoped> implements ComponentScoped

Support inline instantiation of the ComponentScoped annotation.

152.19.9.1 public static final ComponentScoped INSTANCE

Default instance.

152.19.9.2 public Literal()

152.19.10 @FactoryComponent

Identifies a factory component.

Factory components MUST always be ComponentScoped. Applying any other scope will result in a definition error.

See Also Factory Component

Retention RUNTIME

Target TYPE

152.19.10.1 String value default "$"

□ The configuration PID for the configuration of this Component.

The value specifies a configuration PID who's configuration properties are available at injection points in the component.

A special string ("$") can be used to specify the name of the component as a configuration PID. The CDI_COMPONENT_NAME constant holds this special string.

For example:

```
@FactoryPID(CDI_COMPONENT_NAME)
```

152.19.11 public static final class FactoryComponent.Literal extends AnnotationLiteral<FactoryComponent> implements FactoryComponent

Support inline instantiation of the FactoryComponent annotation.

152.19.11.1 public static final FactoryComponent.Literal of(String pid)

pid the factory configuration pid

Returns an instance of FactoryComponent

152.19.11.2 public String value()

152.19.12 @MinimumCardinality

Annotation used in conjunction with Reference to specify the minimum cardinality reference property.

Specifying the MinimumCardinality annotation with the value of 0 on a unary reference is a definition error.

Retention RUNTIME

Target FIELD, PARAMETER

152.19.12.1 int value default 1

□ The minimum cardinality of the reference.

The value must be a positive integer.

For example:

```
@MinimumCardinality(3)
```

152.19.13 public static final class MinimumCardinality.Literal extends AnnotationLiteral<MinimumCardinality> implements MinimumCardinality

Support inline instantiation of the MinimumCardinality annotation.

152.19.13.1 public static final MinimumCardinality.Literal of(int value)

value the minimum cardinality

Returns an instance of MinimumCardinality

152.19.13.2 public int value()

152.19.14 @PID

Annotation used in collaboration with ComponentScoped to specify singleton configurations and their policy.

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.19.14.1 String value default "$"

□ The configuration PID for the configuration of this Component.

The value specifies a configuration PID who's configuration properties are available at injection points in the component.

A special string ("$") can be used to specify the name of the component as a configuration PID. The CDI_COMPONENT_NAME constant holds this special string.

For example:

```
@PID(CDI_COMPONENT_NAME)
```

152.19.14.2 **ConfigurationPolicy policy default OPTIONAL**

□ The configuration policy associated with this PID.

Controls how the configuration must be satisfied depending on the presence and type of a corresponding Configuration object in the OSGi Configuration Admin service. Corresponding configuration is a Configuration object where the PID is equal to value.

If not specified, the configuration is not required.

152.19.15 public static final class PID.Literal extends AnnotationLiteral<PID> implements PID

Support inline instantiation of the PID annotation.

152.19.15.1 **public static final PID.Literal of(String pid, ConfigurationPolicy policy)**

pid the configuration pid

policy the policy of the configuration

Returns an instance of PID

152.19.15.2 **public ConfigurationPolicy policy()**

152.19.15.3 **public String value()**

152.19.16 @PIDs

Annotation used in conjunction with ComponentScoped in order to associate configurations with the component bean.

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.19.16.1 **PID[] value**

□ The set of ordered configurations available to the component.

152.19.17 public static final class PIDs.Literal extends AnnotationLiteral<PIDs> implements PIDs

Support inline instantiation of the PIDs annotation.

152.19.17.1 **public static PIDs of(PID[] pids)**

pids array of PID

Returns an instance of PIDs

152.19.17.2 **public PID[] value()**

152.19.18 @PrototypeRequired

Used with @Reference, BindService, BindBeanServiceObjects and BindServiceReference to indicate that the service must be `service.scope=prototype`.

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.19.19 public static final class PrototypeRequired.Literal extends AnnotationLiteral<PrototypeRequired> implements PrototypeRequired

Support inline instantiation of the PrototypeRequired annotation.

152.19.19.1 public static final PrototypeRequired INSTANCE

Default instance

152.19.19.2 public Literal()

152.19.20 @Reference

Annotation used on injection points informing the CDI container that the injection should apply a service obtained from the OSGi registry.

*

See Also Reference Annotation

Retention RUNTIME

Target FIELD, PARAMETER

152.19.20.1 Class<?> value default Object.class

- Specify the type of the service for this reference.

If not specified, the type of the service for this reference is derived from the injection point type.

If a value is specified it must be type compatible with (assignable to) the service type derived from the injection point type, otherwise a definition error will result.

152.19.20.2 String target default ""

- The target property for this reference.

If not specified, no target property is set.

152.19.21 public static final class Reference.Any

A marker type used in Reference.value to indicate that a reference injection point may accept any service type(s).

The injection point service type must be specified as Object.

The value must be specified by itself.

For example:

```
@Inject
@Reference(value = Any.class, target = "(bar=baz)")
List<Object> services;
```

152.19.21.1 public Any()

152.19.22 public static final class Reference.Literal extends AnnotationLiteral<Reference> implements Reference

Support inline instantiation of the Reference annotation.

152.19.22.1 public static final Reference.Literal of(Class<?> service, String target)

service

target

Returns instance of Reference

152.19.22.2 public String target()

152.19.22.3 public Class<?> value()

152.19.23 @Reluctant

Annotation used to indicate that the behavior of the reference should be reluctant. Used in conjunction with @Reference, BindService, BindServiceReference or BindBeanServiceObjects.

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.19.24 public static final class Reluctant.Literal extends AnnotationLiteral<Reluctant> implements Reluctant

Support inline instantiation of the Reluctant annotation.

152.19.24.1 public static final Reluctant INSTANCE

Default instance

152.19.24.2 public Literal()

152.19.25 @RequireCDIExtender

This annotation can be used to require the CDI Component Runtime extender. It can be used directly, or as a meta-annotation.

Retention CLASS

Target TYPE, PACKAGE

152.19.25.1 String[] descriptor default "META-INF/beans.xml"

□ Specify CDI bean descriptor file paths to be searched on the `Bundle-ClassPath`. For example:

```
@RequireCDIExtender(descriptor = "META-INF/beans.xml")
```

Returns CDI bean descriptor file paths.

152.19.25.2 Class<?>[] beans default {}

□ Specify OSGi Beans classes to be used by the CDI container. For example:

```
@RequireCDIExtender(beans = {com.foo.BarImpl.class, com.foo.impl.BazImpl.class})
```

Returns OSGi Beans classes to be used by the CDI container.

152.19.26 @RequireCDIImplementation

This annotation can be used to require the CDI Component Runtime implementation. It can be used directly, or as a meta-annotation.

Retention CLASS

Target TYPE, PACKAGE

152.19.27 @Service

Annotation used to specify that a bean should be published as a service.

The behavior of this annotation depends on it's usage:

- on the bean type - publish the service using all implemented interfaces. If there are no implemented interfaces use the bean class.
- on the bean's type_use(s) - publish the service using the collected interface(s).

Use of `@Service` on both type and type_use will result in a definition error.

Where this annotation is used affects how service scopes are supported:

- @SingleComponent, @FactoryComponent or @Dependent bean - The provided service can be of any scope. The bean can either implement ServiceFactory or PrototypeServiceFactory or use @Bundle or @Prototype to set it's service scope. If none of those options are used the service is a singleton scope service.
- @ApplicationScoped bean - The provided service is a singleton scope service unless the bean implements ServiceFactory or PrototypeServiceFactory. It cannot use @Bundle or @Prototype to set it's service scope. Use of those annotations in this case will result in a definition error.

Retention RUNTIME

Target FIELD, METHOD, TYPE, TYPE_USE

152.19.27.1 Class<?>[] value default {}

□ Override the interfaces under which this service is published.

Returns the service types

152.19.28 public static final class Service.Literal extends AnnotationLiteral<Service> implements Service

Support inline instantiation of the Service annotation.

152.19.28.1 public static final Service.Literal of(Class<?>[] interfaces)

interfaces

Returns instance of Service

152.19.28.2 public Class<?>[] value()

152.19.29 @ServiceInstance

Annotation used on beans, observer methods and observer fields to specify the service scope for the service. Used in conjunction with Service.

Retention RUNTIME

Target TYPE, FIELD, METHOD

152.19.29.1 ServiceScope value default SINGLETON

□ The scope of the service.

152.19.30 public static final class ServiceInstance.Literal extends AnnotationLiteral<ServiceInstance> implements ServiceInstance

Support inline instantiation of the ServiceInstance annotation.

152.19.30.1 public static ServiceInstance.Literal of(ServiceScope type)

type the type of the ServiceInstance

Returns an instance of ServiceInstance

152.19.30.2 public ServiceScope value()

152.19.31 @SingleComponent

Identifies a single component.

Single components MUST always be ComponentScoped. Applying any other scope will result in a definition error.

See Also Single Component

Retention RUNTIME

Target TYPE

152.19.32 public static final class SingleComponent.Literal extends AnnotationLiteral<SingleComponent> implements SingleComponent

Support inline instantiation of the SingleComponent annotation.

152.19.32.1 public static final SingleComponent INSTANCE

Default instance.

152.19.32.2 public Literal()

152.20 org.osgi.service.cdi.propertytypes

Bean Property Types Package Version 1.0.

When used as annotations, bean property types are processed by CCR to generate default component properties, service properties and target filters.

Bundles wishing to use this package at runtime must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.cdi.propertytypes; version="[1.0,2.0)"
```

152.20.1 Summary

- BeanPropertyException - This Runtime Exception is thrown when a Bean Property Type method attempts an invalid component property coercion.
- ExportedService - Bean Property Type for the remote service properties for an exported service.
- ServiceDescription - Bean Property Type for the service.description service property.

- ServiceRanking - Bean Property Type for the service.ranking service property.
- ServiceVendor - Bean Property Type for the service.vendor service property.

152.20.2 public class BeanPropertyException extends RuntimeException

This Runtime Exception is thrown when a Bean Property Type method attempts an invalid component property coercion. For example when the bean property type method Long test(); is applied to a component property "test" of type String.

152.20.2.1 public BeanPropertyException(String message)

message The message for this exception.

□ Create a Bean Property Exception with a message.

152.20.2.2 public BeanPropertyException(String message, Throwable cause)

message The message for this exception.

cause The causing exception.

□ Create a Bean Property Exception with a message and a nested cause.

152.20.3 @ExportedService

Bean Property Type for the remote service properties for an exported service.

This annotation can be used as defined by BeanPropertyType to declare the values of the remote service properties for an exported service.

See Also Bean Property Types, Remote Services Specification

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.20.3.1 Class<?>[] service_exported_interfaces

□ Service property marking the service for export. It defines the interfaces under which the service can be exported.

If an empty array is specified, the property is not added to the component description.

Returns The exported service interfaces.

See Also Constants.SERVICE_EXPORTED_INTERFACES

152.20.3.2 String[] service_exported_configs default {}

□ Service property identifying the configuration types that should be used to export the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The configuration types.

See Also Constants.SERVICE_EXPORTED_CONFIGS

152.20.3.3 String[] service_exported_intents default {}

□ Service property identifying the intents that the distribution provider must implement to distribute the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The intents that the distribution provider must implement to distribute the service.

See Also Constants.SERVICE_EXPORTED_INTENTS

152.20.3.4 String[] service_exported_intents_extra default {}

□ Service property identifying the extra intents that the distribution provider must implement to distribute the service.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The extra intents that the distribution provider must implement to distribute the service.

See Also Constants.SERVICE_EXPORTED_INTENTS_EXTRA

152.20.3.5 String[] service_intents default {}

□ Service property identifying the intents that this service implements.

If an empty array is specified, the default value, the property is not added to the component description.

Returns The intents that the service implements.

See Also Constants.SERVICE_INTENTS

152.20.4 @ServiceDescription

Bean Property Type for the service.description service property.

This annotation can be used as defined by BeanPropertyType to declare the value the Constants.SERVICE_DESCRIPTION service property.

See Also Bean Property Types

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.20.4.1 String value

□ Service property identifying a service's description.

Returns The service description.

See Also Constants.SERVICE_DESCRIPTION

152.20.5 @ServiceRanking

Bean Property Type for the service.ranking service property.

This annotation can be used as defined by BeanPropertyType to declare the value of the Constants.SERVICE_RANKING service property.

See Also Bean Property Types

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.20.5.1 int value

□ Service property identifying a service's ranking.

Returns The service ranking.

See Also Constants.SERVICE_RANKING

152.20.6 @ServiceVendor

Bean Property Type for the service.vendor service property.

This annotation can be used as defined by BeanPropertyType to declare the value of the Constants.SERVICE_VENDOR service property.

See Also Bean Property Types

Retention RUNTIME

Target FIELD, METHOD, PARAMETER, TYPE

152.20.6.1 String value

□ Service property identifying a service's vendor.

Returns The service vendor.

See Also Constants.SERVICE_VENDOR

152.21 org.osgi.service.cdi.reference

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.cdi.annotations; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.cdi.annotations; version="[1.0,1.1)"
```

152.21.1 Summary

- `BeanServiceObjects` - Allows multiple service objects for a service to be obtained.
- `BindBeanServiceObjects` - A bean provided by CCR for binding actions to life cycle events of matching services.
- `BindService` - A bean provided by CCR for binding actions to life cycle events of matching services.
- `BindServiceReference` - A bean provided by CCR for binding actions to life cycle events of matching services.

152.21.2 public interface BeanServiceObjects<S>

<S> Type of Service

Allows multiple service objects for a service to be obtained.

A component instance can receive a `BeanServiceObjects` object via a reference that is typed `BeanServiceObjects`.

For services with prototype scope, multiple service objects for the service can be obtained. For services with singleton or bundle scope, only one, use-counted service object is available.

Any unreleased service objects obtained from this `BeanServiceObjects` object are automatically released by Service Component Runtime when the service becomes unbound.

See Also ServiceObjects

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

152.21.2.1 public S getService()

□ Returns a service object for the associated service.

This method will always return null when the associated service has been become unbound.

Returns A service object for the associated service or null if the service is unbound, the customized service object returned by a ServiceFactory does not implement the classes under which it was registered or the ServiceFactory threw an exception.

Throws IllegalStateException– If the component instance that received this BeanServiceObjects object has been deactivated.

See Also ungetService(Object)

152.21.2.2 public ServiceReference<S> getServiceReference()

□ Returns the ServiceReference for the service associated with this BeanServiceObjects object.

Returns The ServiceReference for the service associated with this BeanServiceObjects object.

152.21.2.3 public void ungetService(S service)

service A service object previously provided by this ReferenceServiceObjects object.

□ Releases a service object for the associated service.

The specified service object must no longer be used and all references to it should be destroyed after calling this method.

Throws IllegalStateException– If the component instance that received this ReferenceServiceObjects object has been deactivated.

IllegalArgumentException– If the specified service object was not provided by this BeanServiceObjects object.

See Also getService()

152.21.3 public interface BindBeanServiceObjects<S>

<S> the service argument type.

A bean provided by CCR for binding actions to life cycle events of matching services.

See Also Reference

Provider Type Consumers of this API must not implement this type

152.21.3.1 public BindBeanServiceObjects<S> adding(Consumer<BeanServiceObjects<S>> action)

action the action, whose argument is the Bean Service Objects, to subscribe to the *adding* service event

□ Subscribe an action to the *adding* service event.

Only the last *adding* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.21.3.2 public void bind()

□ The bind terminal operation is required to instruct CCR that all the bind actions have been specified, otherwise bind actions will never be called by CCR.

Calling bind again has no effect.

152.21.3.3 public BindBeanServiceObjects<S> modified(Consumer<BeanServiceObjects<S>> action)

action the action, whose argument is the Bean Service Objects, to subscribe to the *modified* service event

□ Subscribe an action to the *modified* service event.

Only the last *modified* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.3.4 public BindBeanServiceObjects<S> removed(Consumer<BeanServiceObjects<S>> action)

action the action, whose argument is the Bean Service Objects, to subscribe to the *removed* service event

□ Subscribe an action to the *removed* service event.

Only the last *removed* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4 public interface BindService<S>

<S> the service argument type.

A bean provided by CCR for binding actions to life cycle events of matching services.

See Also Reference

Provider Type Consumers of this API must not implement this type

152.21.4.1 public BindService<S> adding(Consumer<S> action)

action the action, whose argument is the service instance, to subscribe to the *adding* service event

□ Subscribe an action to the *adding* service event.

Only the last *adding* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4.2 public BindService<S> adding(BiConsumer<S, Map<String, Object>> action)

action the action, whose arguments are the service instance and the Map<String, Object> of service properties, to subscribe to the *adding* service event

□ Subscribe an action to the *adding* service event.

Only the last *adding* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4.3 public void bind()

□ The bind terminal operation is required to instruct CCR that all the bind actions have been specified, otherwise bind actions will never be called by CCR.

Calling bind again has no effect.

152.21.4.4 public BindService<S> modified(Consumer<S> action)

action the action, whose argument is the service instance, to subscribe to the *modified* service event

□ Subscribe an action to the *modified* service event.

Only the last *modified* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4.5 public BindService<S> modified(BiConsumer<S, Map<String, Object>> action)

action the action, whose arguments are the service instance and the Map<String, Object> of service properties, to subscribe to the *modified* service event

□ Subscribe an action to the *modified* service event.

Only the last *modified* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4.6 public BindService<S> removed(Consumer<S> action)

action the action, whose argument is the service instance, to subscribe to the *removed* service event

□ Subscribe an action to the *removed* service event.

Only the last *removed* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.4.7 public BindService<S> removed(BiConsumer<S, Map<String, Object>> action)

action the action, whose arguments are the service instance and the Map<String, Object> of service properties, to subscribe to the *removed* service event

□ Subscribe an action to the *removed* service event.

Only the last *removed* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.5 public interface BindServiceReference<S>

<S> the service argument type.

A bean provided by CCR for binding actions to life cycle events of matching services.

See Also Reference

Provider Type Consumers of this API must not implement this type

152.21.5.1 public BindServiceReference<S> adding(Consumer<ServiceReference<S>> action)

action the action, whose argument is the service reference, to subscribe to the *adding* service event

□ Subscribe an action to the *adding* service event.

Only the last *adding* action is used.

Returns self

Throws IllegalStateException – when called after bind

152.21.5.2 public BindServiceReference<S> adding(BiConsumer<ServiceReference<S>, S> action)

action the action, whose arguments are the service reference and the service object, to subscribe to the *adding* service event

□ Subscribe an action to the *adding* service event.

Only the last *adding* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.21.5.3 public void bind()

□ The bind terminal operation is required to instruct CCR that all the bind actions have been specified, otherwise bind actions will never be called by CCR.

Calling bind again has no effect.

152.21.5.4 public BindServiceReference<S> modified(Consumer<ServiceReference<S>> action)

action the action, whose argument is the service reference, to subscribe to the *modified* service event

□ Subscribe an action to the *modified* service event.

Only the last *modified* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.21.5.5 public BindServiceReference<S> modified(BiConsumer<ServiceReference<S>, S> action)

action the action, whose arguments are the service reference and the service object, to subscribe to the *modified* service event

□ Subscribe an action to the *modified* service event.

Only the last *modified* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.21.5.6 public BindServiceReference<S> removed(Consumer<ServiceReference<S>> action)

action the action, whose argument is the service reference, to subscribe to the *removed* service event

□ Subscribe an action to the *removed* service event.

Only the last *removed* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.21.5.7 public BindServiceReference<S> removed(BiConsumer<ServiceReference<S>, S> action)

action the action, whose arguments are the service reference and the service object, to subscribe to the *removed* service event

□ Subscribe an action to the *removed* service event.

Only the last *removed* action is used.

Returns self

Throws IllegalStateException– when called after bind

152.22 org.osgi.service.cdi.runtime

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.cdi; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.cdi; version="[1.0,1.1)"`

152.22.1 Summary

- `CDIComponentRuntime` - The CDIComponentRuntime service represents the actor that manages the CDI containers and their life cycle.

152.22.2 public interface CDIComponentRuntime

The CDIComponentRuntime service represents the actor that manages the CDI containers and their life cycle. The CDIComponentRuntime service allows introspection of the managed CDI containers.

This service must be registered with a Constants.SERVICE_CHANGECOUNT service property that must be updated each time any of the DTOs available from this service change.

Access to this service requires the `ServicePermission[CDIComponentRuntime, GET]` permission. It is intended that only administrative bundles should be granted this permission to limit access to the potentially intrusive methods provided by this service.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

152.22.2.1 public Collection<ContainerDTO> getContainerDTOs(Bundle... bundles)

bundles The bundles who's container description snapshots are to be returned. Specifying no bundles, or the equivalent of an empty `Bundle` array, will return the container descriptions of all active bundles that define a container.

□ Returns a collection of container description snapshots for a set of bundles.

Returns A set of descriptions of the container of the specified `bundles`. Only bundles that have an associated container are included. If a bundle is listed multiple times in `bundles` only one ContainerDTO is returned. Returns an empty collection if no CDI containers are found.

152.22.2.2 public ContainerTemplateDTO getContainerTemplateDTO(Bundle bundle)

bundle The bundle defining a container. Must not be `null` and must be active.

□ Returns the ContainerTemplateDTO for the specified bundle

Returns The container template for of the specified bundle or `null` if it does not have an associated container.

152.23 org.osgi.service.cdi.runtime.dto

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.cdi.dto; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.cdi.dto; version="[1.0,1.1)"`

152.23.1 Summary

- ActivationDTO - A snapshot of the runtime state of a component activation.
- ComponentDTO - A snapshot of the runtime state of a component.
- ComponentInstanceDTO - A snapshot of the runtime state of a component.
- ConfigurationDTO - A snapshot of the runtime state of a component factory configuration dependency
- ContainerDTO - A snapshot of the runtime state of a CDI container
- ExtensionDTO - A snapshot of the runtime state of an javax.enterprise.inject.spi.Extension dependency required by this CDI container.
- ReferenceDTO - A snapshot of the runtime state of a component reference dependency

152.23.2 public class ActivationDTO extends DTO

A snapshot of the runtime state of a component activation.

Concurrency Not Thread-safe

152.23.2.1 public List<String> errors

The list of errors which occurred during initialization. An empty list means there were no errors.

Must not be null.

152.23.2.2 public ServiceReferenceDTO service

The service this activation may have registered.

Must not be null if template.serviceClasses is not empty.

152.23.2.3 public ActivationTemplateDTO template

The template describing this activation.

Must not be null

152.23.2.4 public ActivationDTO()

152.23.3 public class ComponentDTO extends DTO

A snapshot of the runtime state of a component.

Concurrency Not Thread-safe

152.23.3.1 public boolean enabled

Indicates if the component is enabled. The default is true.

A setting of false on the *container component* results in all components in the bundle being disabled.

152.23.3.2 public List<ComponentInstanceDTO> instances

The component instances created by this component.

- When template is of type ComponentType.CONTAINER - there will be 1 ComponentInstanceDTO
- When template is of type ComponentType.SINGLE - there will be 1 ComponentInstanceDTO
- When template is of type ComponentType.FACTORY - there will be one ComponentInstanceDTO for every factory configuration object associated with the factory PID of the component.

Must not be null

152.23.3.3 public ComponentTemplateDTO template

The template of this component.

Must not be null

152.23.3.4 public ComponentDTO()

152.23.4 public class ComponentInstanceDTO extends DTO

A snapshot of the runtime state of a component.

Concurrency Not Thread-safe

152.23.4.1 public List<ActivationDTO> activations

The activations of the component.

Must not be null.

152.23.4.2 public List<ConfigurationDTO> configurations

The configuration dependencies of this component.

Must not be null.

152.23.4.3 public Map<String, Object> properties

The resolved configuration properties for the component.

Contains the merger of all consumed configurations merged in the order of configurations.

All configuration dependencies are satisfied when not null.

152.23.4.4 public List<ReferenceDTO> references

The service dependencies of the component.

Can be empty when the component has no reference dependencies.

The component instance is satisfied when the sum of ReferenceDTO.minimumCardinality equals the size of ReferenceDTO.matches for each value.

Must not be null.

152.23.4.5 public ComponentInstanceDTO()

152.23.5 public class ConfigurationDTO extends DTO

A snapshot of the runtime state of a component factory configuration dependency

Concurrency Not Thread-safe

152.23.5.1 public Map<String, Object> properties

The properties of this configuration.

The configuration dependency is satisfied when not null.

152.23.5.2 public ConfigurationTemplateDTO template

The template of this configuration dependency

Must never be null

152.23.5.3 public ConfigurationDTO()

152.23.6 public class ContainerDTO extends DTO

A snapshot of the runtime state of a CDI container

Concurrency Not Thread-safe

152.23.6.1 public BundleDTO bundle

The bundle declaring the CDI container.

Must not be 0.

152.23.6.2 public long changeCount

The change count of the container at the time this DTO was created

Must not be 0.

152.23.6.3 public List<ComponentDTO> components

The components defined by this CDI container.

Must not be null. The list always contains at least one element representing the container component. See *Container Component*.

152.23.6.4 public List<String> errors

The list of errors reported during attempted initialization of the container instance.

152.23.6.5 public List<ExtensionDTO> extensions

The extension dependencies of this CDI container.

Must not be null.

152.23.6.6 public ContainerTemplateDTO template

The template of this Container DTO.

Must not be null.

152.23.6.7 public ContainerDTO()

152.23.7 public class ExtensionDTO extends DTO

A snapshot of the runtime state of an javax.enterprise.inject.spi.Extension dependency required by this CDI container.

Concurrency Not Thread-safe

152.23.7.1 public ServiceReferenceDTO service

The service reference of the extension.

The extension dependency is satisfied when not null.

152.23.7.2 public ExtensionTemplateDTO template

The template of this extension dependency.

Must not be null

152.23.7.3 public ExtensionDTO()

152.23.8 public class ReferenceDTO extends DTO

A snapshot of the runtime state of a component reference dependency

Concurrency Not Thread-safe

152.23.8.1 public List<ServiceReferenceDTO> matches

The list of service references that match this reference.

Must not be `null`

Can be empty when there are no matching services.

This dependency is satisfied when minimumCardinality <= `matches.size() <=` MaximumCardinality.toInt() where the maximum cardinality can be obtained from the associated ReferenceTemplateDTO.

152.23.8.2 public int minimumCardinality

The runtime minimum cardinality of the dependency.

- If template.maximumCardinality is ONE the value must be either 0 or 1.
- If template.maximumCardinality is MANY the value must be from 0 to Integer.MAX_VALUE.

152.23.8.3 public String targetFilter

Indicates the runtime target filter used in addition to the template.serviceType to match services.

152.23.8.4 public ReferenceTemplateDTO template

The template of this reference.

Must not be `null`

152.23.8.5 public ReferenceDTO()

152.24 org.osgi.service.cdi.runtime.dto.template

CDI Integration Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.cdi.dto.model; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.cdi.dto.model; version="[1.0,1.1)"`

152.24.1 Summary

- `ActivationTemplateDTO` - Activations represent either immediate instances or service objects produced by component instances.
- `ComponentTemplateDTO` - A static description of a CDI component.

- ConfigurationTemplateDTO - A description of a configuration dependency of a component The content of this DTO is resolved form metadata at initialization time and remains the same between the CDI bundle restarts.
- ContainerTemplateDTO - Description of a CDI container.
- ExtensionTemplateDTO - Models an extension dependency of the ContainerDTO
- ReferenceTemplateDTO - A description of a reference dependency of a component

152.24.2 public class ActivationTemplateDTO extends DTO

Activations represent either immediate instances or service objects produced by component instances.

The content of this DTO is resolved form metadata at initialization time and remains the same between the CDI bundle restarts.

Concurrency Not Thread-safe

152.24.2.1 public Map<String, Object> properties

The default properties for activations which represent container component services. This will never be populated for single or factory components.

These are merged (and possibly replaced) with runtime properties.

Must not be null. May be empty if no default properties are provided.

152.24.2.2 public ServiceScope scope

The ServiceScope of this activation

Must not be null.

152.24.2.3 public List<String> serviceClasses

Describes the set of fully qualified names of the interfaces/classes under which this activation will publish and OSGi service

Must not be null. An empty array indicated this activation will not publish an OSGi service

152.24.2.4 public ActivationTemplateDTO()

152.24.3 public class ComponentTemplateDTO extends DTO

A static description of a CDI component.

At runtime it is spit between a ComponentInstanceDTO which handles the resolution of the configurations, references and the creation of ComponentInstanceDTO instances and one or more ComponentInstanceDTO instances, which handle the resolution of references and the creation of activations.

Concurrency Not Thread-safe

152.24.3.1 public List<ActivationTemplateDTO> activations

The activations associated with the component.

Must not be null.

152.24.3.2 public List<String> beans

The set of beans that make up the component.

Must not be null.

152.24.3.3 public List<ConfigurationTemplateDTO> configurations

The configuration dependencies of this component.

There is always at least one default singleton configuration.

May contain at most one factory configuration.

Must not be null.

152.24.3.4 public String name

A name unique within the container.

Must not be null.

152.24.3.5 public Map<String, Object> properties

The default component properties.

These are merged (and possibly replaced) with runtime properties.

Must not be null. May be empty if no default properties are provided.

152.24.3.6 public List<ReferenceTemplateDTO> references

The service dependencies of the component.

The list will be empty if there are no service dependencies.

Must not be null.

152.24.3.7 public ComponentType type

The type of the component.

Must not be null.

152.24.3.8 public ComponentTemplateDTO()

152.24.4 public class ConfigurationTemplateDTO extends DTO

A description of a configuration dependency of a component The content of this DTO is resolved form metadata at initialization time and remains the same between the CDI bundle restarts.

Concurrency Not Thread-safe

152.24.4.1 public MaximumCardinality maximumCardinality

The maximum cardinality of the configuration dependency.

- When MaximumCardinality.ONE this is a singleton configuration dependency.
- When MaximumCardinality.MANY this is a factory configuration dependency.

Must not be null.

152.24.4.2 public String pid

The PID of the tracked configuration object(s).

Must not be null.

152.24.4.3 public ConfigurationPolicy policy

The policy for the configuration dependency.

Must not be null.

152.24.4.4 public ConfigurationTemplateDTO()

152.24.5 public class ContainerTemplateDTO extends DTO

Description of a CDI container.

Concurrency Not Thread-safe

152.24.5.1 public List<ComponentTemplateDTO> components

The components defined in this CDI container.

Must not be null

Has at lest one element for the container component. See *Container Component*.

152.24.5.2 public List<ExtensionTemplateDTO> extensions

The extension dependencies of this CDI container.

Must not be null

May be empty if the CDI container does not require CDI extensions.

152.24.5.3 public String id

The id of the CDI container.

152.24.5.4 public ContainerTemplateDTO()

152.24.6 public class ExtensionTemplateDTO extends DTO

Models an extension dependency of the ContainerDTO

Concurrency Not Thread-safe

152.24.6.1 public String serviceFilter

The service filter used for finding the extension service.

The value must be associated to the osgi.cdi extender requirement whose 'extension' attribute contains a value equal to serviceFilter.

Must not be null.

152.24.6.2 public ExtensionTemplateDTO()

152.24.7 public class ReferenceTemplateDTO extends DTO

A description of a reference dependency of a component

The content of this DTO is resolved form metadata at initialization time and remains the same between the CDI bundle restarts.

Concurrency Not Thread-safe

152.24.7.1 public MaximumCardinality maximumCardinality

The maximum cardinality of the reference.

152.24.7.2 public int minimumCardinality

The minimum cardinality of the reference.

Contains the minimum cardinality statically resolved from the CDI bundle metadata. The minimum cardinality can be replaced by configuration at runtime.

- If maximumCardinality is ONE the value must be either 0 or 1.
- If maximumCardinality is MANY the value must be from 0 to Integer.MAX_VALUE.

152.24.7.3 public String name

A unique within the container and persistent across reboots identified for this activation

The value must not be null. The value must be equal to the reference name.

152.24.7.4 public ReferencePolicy policy

Indicates if the reference is dynamic or static in nature.

152.24.7.5 public ReferencePolicyOption policyOption

Indicates if the reference is greedy or reluctant in nature.

152.24.7.6 public String serviceType

Indicates the type of service matched by the reference.

The value must not be null.

152.24.7.7 public String targetFilter

Indicates a target filter used in addition to the serviceType to match services.

Contains the target filter resolved from the CDI bundle metadata. The filter can be replaced by configuration at runtime.

152.24.7.8 public ReferenceTemplateDTO()

152.25 References

[1] *CDI*
https://www.cdi-spec.org/

[2] *CDI 2.0*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html

[3] *unproxyable bean types*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#unproxyable

[4] *Default bean discovery mode*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#default_bean_discovery

[5] *Exclude filters*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#exclude_filters

[6] *Packaging and deployment*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#packaging_deployment

[7] *Typesafe Resolution*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#typesafe_resolution

[8] *Scopes and contexts*

https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#contexts

[9] *Pseudo-scope*
https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html#normal_scope

[10] *@PostConstruct*
https://jakarta.ee/specifications/annotations/1.3/apidocs/javax/annotation/postconstruct

[11] *General Syntax Definitions*
OSGi Core, General Syntax Definitions

[12] *Filter Syntax*
OSGi Core, Filter Syntax

[13] *Dependency Injection for Java*
https://jakarta.ee/specifications/dependency-injection/

[14] *Java Transaction API*
https://jakarta.ee/specifications/transactions/

[15] *Portable Java Contract*
https://docs.osgi.org/reference/portable-java-contracts.html

[16] *The Java Language Specification, Java SE 8 Edition*
https://docs.oracle.com/javase/specs/jls/se8/html/index.html

153 Service Layer API for oneM2M™

Version 1.0

153.1 Introduction of oneM2M

oneM2M™ is a standard for IoT platform, which is standardized by oneM2M partnership project. oneM2M defines set of functionalities that are commonly used in IoT applications, which is called Common Services Function (CSF). The implementation of the CSF is provided by Communication Service Entity (CSE). oneM2M also defines the interface to use the CSF with REST oriented API that consist of limited types of operation (CREATE, RETRIEVE, UPDATE, DELETE, NOTIFY) on many types of resources. Applications of oneM2M use the interface to communicate with CSEs. In a system managed by a single service provider, multiple CSEs can exist and they form tree structure. The root CSE is called Infrastructure Node CSE (IN-CSE). Each application connects to one of CSEs in the system. CSEs have routing capability and application can send request to any CSEs in the system through the directly-connected CSE.

One of characteristic aspects of oneM2M is to allow multiple protocols and serialization formats for messages. Currently specified protocols are HTTP, CoAP, MQTT and WebSocket, and specified serialization are XML, JSON and CBOR (Concise Binary Object Representation). To make specification coherent, oneM2M specifications are separated into abstract level and concrete level. As abstract level, TS-0001 defines the oneM2M architecuture and resource types and TS-0004 defines data procedures and data structures. As concrete level, TS-0008 , TS-0009 , TS-0010 , and TS-0020 define concrete protocol which are mappoed to model of the abstract level. Here, the interface defined in abstract level, which independent on concrete protocols, is regarded as oneM2M Service Layer.

oneM2M Partners Type 1 (ARIB, ATIS, CCSA, ETSI, TIA, TSDSI, TTA, TTC) register oneM2M trademarks and logos in their respective jurisdictions.

153.2 Application Portability Problem of oneM2M

One of potential problems is application portability. oneM2M specifies protocol based interfaces, but doesn't specify a programming level API. Without a standardized API, application program tends to be built tightly coupled with the libraries handling the communication method (combination of protocol and serialization) that is initially intend to use. In that case it would be hard to operate the application in another environment where different communication method is used; basically it is required to modify the application drastically. oneM2M could introduce segmentation of ecosystem within oneM2M compliant applications due to the lack of application portability.

153.3 Introduction of Service Layer API for oneM2M

This chapter provides interface to oneM2M applications for communicating communicate CSE at Service Layer of oneM2M. The providing API is protocol and serialization agnostic for preventing the problem above. Once application developer write code, it can be run in other environment where different communication method is used.

Another benefit of the service is reduction of computational resources, typically latency of execution in a certain cases, where both application and CSE is implemented on OSGi framework. In that case, it is possible to reduce executiontime for serialization/deserialization of data, context-switch of applications, compared to the case where they communicates with a certain communication protocol.

153.4 Essentials

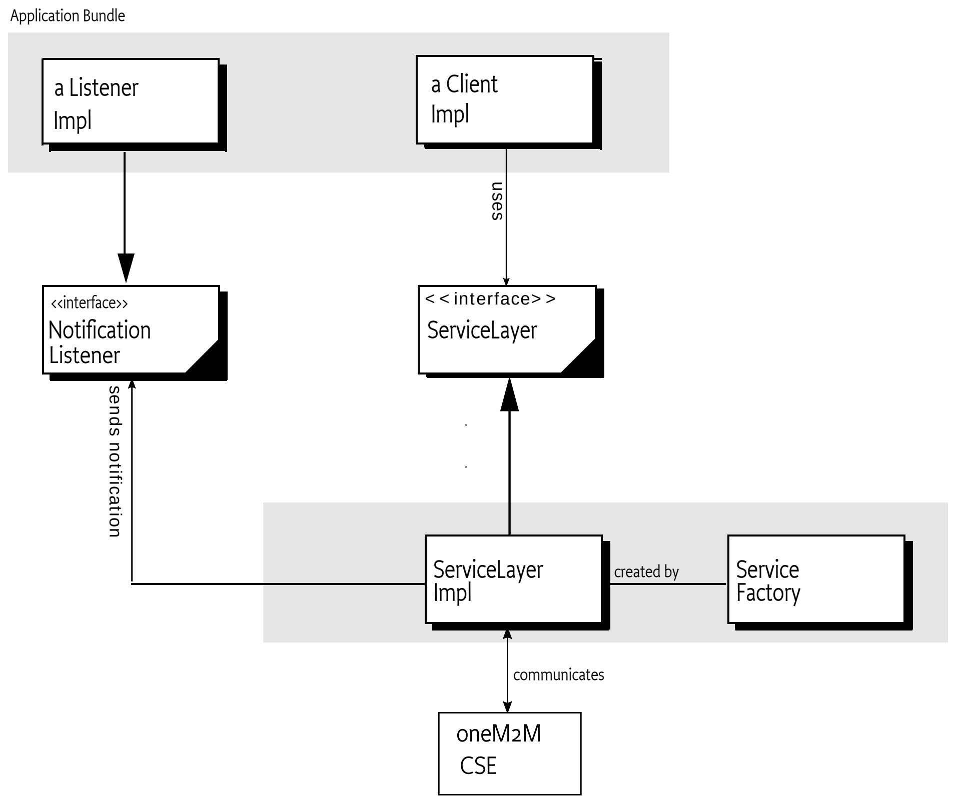
- *Protocol Agnostic* - API is independent on protocol that are used in communications. oneM2M specifies multiple protocols, which are HTTP, CoAP, MQTT and WebSocket. conversion operations.
- *Serialization Agnostic* - API is independent on serialization that are actually used in communications. oneM2M specifies multiple serializations, which are XML, JSON and CBOR .
- *Support of synchoronous and asynchronous method call* - API allows both of calling manners.
- *Use of Data Transfer Object (DTO)* - DTO is used as parameters passing on API. Since oneM2M defines many types, concrete DTOs are specified for the higher level structure, and for lower structures generic DTO is used.
- *Low level and high level operations* - API allows for applications to use both low level operation and high level operation, where low level operation allows all possible oneM2M operations and high level operation allows resource level operations, which are create, retrieve, update, and delete.

153.5 Entities

The following entities are used in this specification:

- *Application Bundle* - Application, which use oneM2M CSE's capability. This specification assumes that an application bundle consists an oneM2M application.
- *ServiceLayer* - This is the API used by oneM2M applications.
- *NotificationListener* - Listener Interface, which is necessary to implement by oneM2M applications, when then need to received notifications.
- *ServiceLayer Implementation Bundle* - Bundle providing implementation of ServiceLayer and its ServiceFactory.
- *oneM2M CSE* - oneM2M's Server. It may exist remotely or locally.

Figure 153.1 *Entity overview of Service Layer API for oneM2M*



153.6 oneM2M ServiceLayer

oneM2M ServiceLayer is the interface used by an application for sending request and get response as return method. It contains low level API and high level API.

request() method allows very raw data type access and it enables all possible message exchanges among oneM2M entities. This is called the low level API. The method takes requestPrimitive as an argument and returns responseRequest. It allows all possible operation of oneM2M. For the return type, OSGi Promise (*Promises Specification* on page 1389) is used for allowing synchronous and asynchronous calling manner.

The low level API may be redundant to application developers, because they need to write composition of requestPrimitive and decomposition of responsePrimitive. Following methods allow application developers to develop application with less lines of code. They provides higher level of abstraction; operation level of resource such as create, retrieve, update, delete, notify and discovery. They cover typical oneM2M operations but do not cover all of possible messages of oneM2M.

Implementation of these high level API automatically inserts 'requestID' and 'from' parameter to RequestDTO.

Following example shows temperature measurement application using container resource and contentInstance resource.

```
ServiceReference<ServiceLayer> sr = bc
```

```
                                .getServiceReference(ServiceLayer.class);
ServiceLayer sl = bc.getService(sr);

ResourceDTO container = new ResourceDTO();
container.resourceType = Constants.RT_contentInstance;
container.resourceName = "temperatureStore";
sl.create("/CSE1/csebase", container).getValue();

ScheduledExecutorService service = Executors
                .newSingleThreadScheduledExecutor();

AtomicInteger count = new AtomicInteger(0);

service.scheduleAtFixedRate(() -> {
        ResourceDTO instance = new ResourceDTO();
        instance.resourceType = Constants.RT_contentInstance;
        instance.resourceName = "instance" + count.getAndIncrement();
        instance.attribute = new HashMap<String,Object>();
        instance.attribute.put("content", measureTemperature());

        sl.create("/CSE1/csebase/temperatureStore", instance);
}, 0, 60, TimeUnit.SECONDS);
```

Following example shows visualizing application of temperature data.

```
ServiceReference<ServiceLayer> sr = (ServiceReference<ServiceLayer>) bc
                .getServiceReference("org.osgi.service.onem2m.ServiceLayer");
ServiceLayer sl = bc.getService(sr);

FilterCriteriaDTO fc = new FilterCriteriaDTO();
fc.createdAfter = "20200101T120000";
fc.createdBefore = "20200101T130000";
List<Integer> resourceTypes = new ArrayList<Integer>();
resourceTypes.add(Constants.RT_contentInstance);
fc.resourceType = resourceTypes;
fc.filterOperation = FilterOperation.AND;

List<String> l = sl.discovery("/CSE1/csebase/temperatureStore", fc)
                .getValue();
List<Pair<String,Double>> renderData = new ArrayList<Pair<String,Double>>();
for (String uri : l) {
        ResourceDTO resource = sl.retrieve(uri).getValue();
        renderData.add(new Pair<String,Double>(resource.creationTime,
                        (Double) resource.attribute.get("content")));
}

renderService(renderData);
```

153.7 NotificationListener

NotificationListener is an interface for receiving oneM2M notification. An application that needs to receive oneM2M notifications must implement the interface and register it to the OSGi registry.

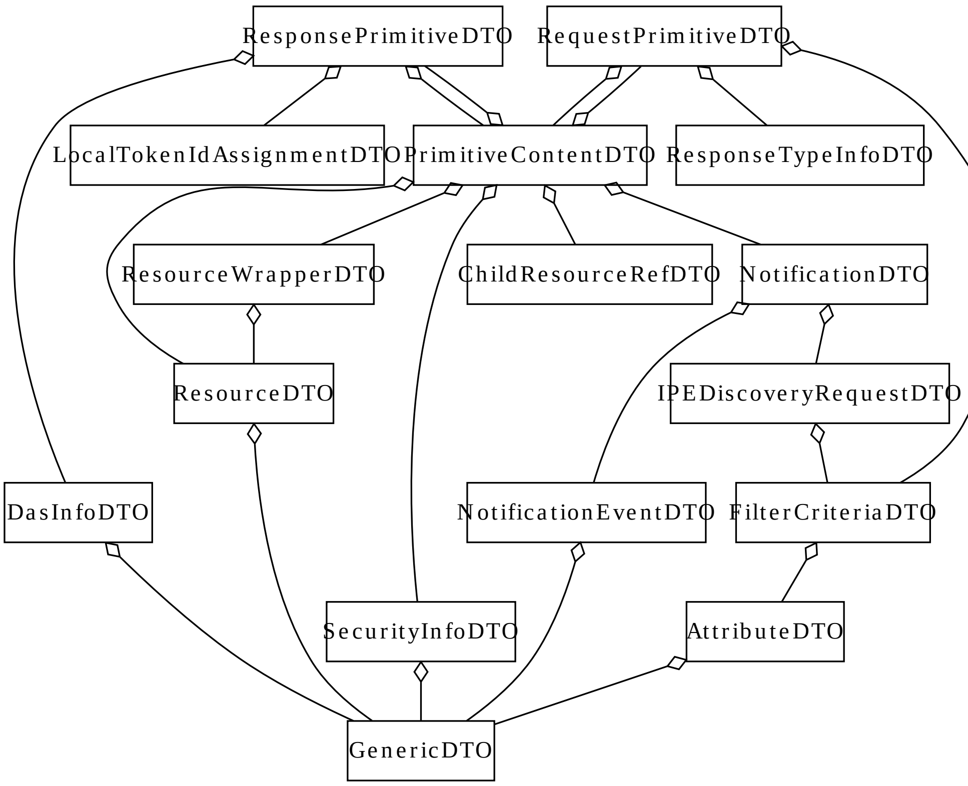
A ServiceLayer Implementation Bundle must call the notify() method of the NotificationListener, when it receives notification from CSE. In notification, target address is designated by AE-ID. The ServiceLayer Implementation Bundle finds the coresponding instance of the NotificationListener by checking its registerer bundle and checking internal mapping table of AE-ID and application bundle.

```
public class MyListener implements NotificationListener {
    public void notified(RequestPrimitiveDTO request){
       NotificationDTO notification = request.content.notification;
       NotificationEventDTO event = notification.notificationEvent;
     Object updatedResource = event.representation;
       NotificationEventType type = event.notificationEventType;
       if( type == NotificationEventType.update_of_resource ){
           // check updated resource, execute some actions.
       }
   }
}

@Activate
public void activate(BundleContext bc) {
    NotificationListener l = new MyListener();
      bc.registerService(NotificationListener.class, l, null);
    }
}
```

153.8 DTO

OSGi DTOs are used for representing data structured passed on the API. Some of the data structures, which are directly referred from API or in small number of hops, are specified with concrete field names. The following figure shows DTOs with concrete field names, and reference relationship of class. Following DTO's rule, instances must not have loop reference relationship.

Figure 153.2 *DTOs representing high level structures*



ResourceDTO represents oneM2M resource. ResourceDTO has both fields with concrete names and a field (named as attribute) for having sub-elements in generic manner. All of fields of the ResourceDTO represent attributes. Most of attributes have a primitive type and part of attributes have structured value. For structured value, if it possible defined concrete DTOs must be used, otherwise GenericDTO must be used.

oneM2M specifies two types of key names for representing name of resources, attributes, and elements of data structure, which are long name and short name. Long name is human readable representation, for example "resourceID", meanwhile short name is compact representation for minimizing network transfer, consist with typically 2-4 alphabetical characters, for example "ri". All field names in concrete DTOs are based on long name. Long name should be used for key names of GenericDTO and attribute names of ResourceDTO.

153.9 Security

Implementation of ServiceLayer may use credentials on behalf of application bundles on the communication with oneM2M CSE. So ServiceLayer Implementation should pass the service reference of ServiceLayer to only the proper application bundle. Use of ServiceFactory is to realize this. Application Bundles should not pass the service reference to other application bundles.

How to configure those credentials is left to developer of ServiceLayer Implementation, and it is out of scope the specification.

153.10 org.osgi.service.onem2m

Service Layer API for oneM2M Specification Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.onem2m; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.onem2m; version="[1.0,1.1)"
```

153.10.1 Summary

- `NotificationListener` - Interface to receive notification from other oneM2M entities.
- `OneM2MException` - General Exception for oneM2M.
- `ServiceLayer` - Primary Interface for an oneM2M application entity to send request and get response to/from other oneM2M entity.

153.10.2 public interface NotificationListener

Interface to receive notification from other oneM2M entities.

Application that receives notification must implement this interface and register to OSGi service registry. No service property is required.

153.10.2.1 public void notified(RequestPrimitiveDTO request)

request request primitive

□ receive notification.

153.10.3 public class OneM2MException extends IOException

General Exception for oneM2M.

153.10.3.1 public OneM2MException(String message, int errorCode)

message The exception message.

errorCode The exception error code.

□ Construct a OneM2MException with a message and an error code.

153.10.3.2 public int getErrorCode()

□ Return the error code for the exception.

Returns The error code for the exception.

153.10.4 public interface ServiceLayer

Primary Interface for an oneM2M application entity to send request and get response to/from other oneM2M entity.

It contains low level API and high level API. The only low level method is request() and other methods are categorized as high level API.

Provider Type Consumers of this API must not implement this type

153.10.4.1 public Promise<ResourceDTO> create(String uri, ResourceDTO resource)

uri URI for parent resource of the resource being created.

resource resource data

□ create resource

The create() method is a method to create new resource under specified uri. The second argument resource is expression of resource to be generated. The resourceType field of the resourceDTO must be assigned. For other fields depends on resource type. Section 7.4 of TS-00004 specifies the optionalities of the fields.

Returns Promise of created resource

153.10.4.2 public Promise<Boolean> delete(String uri)

uri target URI for deleting resource

□ delete resource

delete resource on the URI specified by uri argument.

Returns promise of execution status

153.10.4.3 public Promise<List<String>> discovery(String uri, FilterCriteriaDTO fc)

uri URI for resource tree to start the search

fc filter criteria selecting resources

□ find resources with filter condition specified in fc argument.

Discovery Result Type is kept as blank and default value of target CSE is used for the parameter.

Returns list of URIs matching the condition specified in fc

153.10.4.4 public Promise<List<String>> discovery(String uri, FilterCriteriaDTO fc, RequestPrimitiveDTO.DesiredIdentifierResultType drt)

uri URI for resource tree to start the search

fc filter criteria

drt Discovery Result Type (structured/unstructured)

□ find resources with filter condition specified in fc argument.

With this method application can specify desired identifier in result

Returns list of URIs matching the condition specified in fc

153.10.4.5 public Promise<Boolean> notify(String uri, NotificationDTO notification)

uri uri of destination

notification content of notification

□ send notification

Returns Promise of notification execution status

153.10.4.6 public Promise<ResponsePrimitiveDTO> request(RequestPrimitiveDTO request)

request request primitive

□ send a request and receive response.

This method allows very raw data type access and it enables all possible message exchanges among oneM2M entities. This is called the low level API. This method allows all possible operation of oneM2M. For the return type, OSGi Promise is used for allowing synchronous and asynchronous calling manner.

Returns promise of ResponseDTO.

153.10.4.7 public Promise<ResourceDTO> retrieve(String uri)

uri URI for retrieving resource

□ retrieve resource

retrieve resource on URI specified by uri argument. This method retrieve all attributes of the resource.

Returns retrieved resource data

153.10.4.8 public Promise<ResourceDTO> retrieve(String uri, List<String> targetAttributes)

uri URI for retrieving resource

targetAttributes names of the target attribute

□ retrieve resource with selected attributes.

retrieve resource on URI specified by uri argument. This method retrieve selected attributes by targetAttributes argument. The retrieve() methods are methods to retrieve resource on URI specified by uri argument.

Returns retrieved resource data

153.10.4.9 public Promise<ResourceDTO> update(String uri, ResourceDTO resource)

uri URI for updating resource

resource data resource

□ update resource

The update() method is a method to update resource on the URI specified by uri argument. The resource argument holds attributes to be updated. Attributes not to be updated shall not included in the argument.

Returns updated resource

153.11 org.osgi.service.onem2m.dto

Service Layer Data Transfer Objects for oneM2M Specification Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.onem2m.dto; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.onem2m.dto; version="[1.0,1.1)"`

153.11.1 Summary

- `AttributeDTO` - DTO expresses Attribute.
- `ChildResourceRefDTO` - DTO expresses ChildResourceRef.
- `Constants` - This class defines constants for resource types.
- `DasInfoDTO` - DTO expresses DasInfo.
- `FilterCriteriaDTO` - DTO expresses FilterCriteria.
- `FilterCriteriaDTO.FilterOperation` - Enum FilterOperation
- `FilterCriteriaDTO.FilterUsage` - Enum FilterUsage
- `GenericDTO` - GenericDTO expresses miscellaneous data structures of oneM2M.
- `IPEDiscoveryRequestDTO` - IPEDiscoveryRequestDTO is an element of NotificationEventDTO
- `LocalTokenIdAssignmentDTO` - DTO expresses LocalTokenIdAssignment.
- `NotificationDTO` - DTO expresses Notification.
- `NotificationEventDTO` - DTO expresses NotificationEventDTO
- `NotificationEventDTO.NotificationEventType` - NotificationEventType
- `PrimitiveContentDTO` - DTO expresses Primitive Content.
- `ReleaseVersion` - enum expresses oneM2M specification version.
- `RequestPrimitiveDTO` - DTO expresses Request Primitive.
- `RequestPrimitiveDTO.DesiredIdentifierResultType` - Enum for DesiredIdentifierResultType
- `RequestPrimitiveDTO.Operation` - enum type for Operation
- `RequestPrimitiveDTO.ResultContent` - enum type for Result Content
- `ResourceDTO` - DTO expresses Resource.
- `ResourceWrapperDTO` - DTO expresses ResourceWrapper.
- `ResponsePrimitiveDTO` - DTO expresses Response Primitive.
- `ResponsePrimitiveDTO.ContentStatus` - Enum ContentStatus
- `ResponseTypeInfoDTO` - DTO expresses ResponseTypeInfo
- `ResponseTypeInfoDTO.ResponseType` - enum ResponseType
- `SecurityInfoDTO` - DTO expresses Security Info.
- `SecurityInfoDTO.SecurityInfoType` - Enum SecurityInfoType

153.11.2 public class AttributeDTO extends DTO

DTO expresses Attribute.

This class is typically used in FilterCriteriaDTO for expressing matching condition.

See Also oneM2M TS-0004 6.3.5.9 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.2.1 public String name

Attribute name

153.11.2.2 public Object value

Supposed value of the attribute

153.11.2.3 public AttributeDTO()

153.11.3 public class ChildResourceRefDTO extends DTO

DTO expresses ChildResourceRef.

See Also oneM2M TS-0004 6.3.5.29 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD childResourceRef [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L885-893]

Concurrency Not Thread-safe

153.11.3.1 public String name

name of the child resource pointed to by the URI

153.11.3.2 public String specializationID

resource type specialization of the child resource pointed to by the URI in case type represents a flexContainer. This is an optional field.

153.11.3.3 public Integer type

resourceType of the child resource pointed to by the URI

153.11.3.4 public String uri

URI to the child resource.

153.11.3.5 public ChildResourceRefDTO()

153.11.4 public final class Constants

This class defines constants for resource types.

See Also oneM2M TS-0004 6.3.4.2.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.4.1 public static final int RT_accessControlPolicy = 1

resource type for accessControlPolicy

153.11.4.2 public static final int RT_accessControlPolicyAnnc = 10001

resource type for accessControlPolicyAnnc

153.11.4.3 public static final int RT_AE = 2

resource type for AE

153.11.4.4 public static final int RT_AEAnnc = 10002

resource type for AEAnnc

153.11.4.5 public static final int RT_AEContactList = 43

resource type for AEContactList

153.11.4.6 public static final int RT_AEContactListPerCSE = 44

resource type for AEContactListPerCSE

153.11.4.7 public static final int RT_authorizationDecision = 35

resource type for authorizationDecision

153.11.4.8 **public static final int RT_authorizationInformation = 37**

resource type for authorizationInformation

153.11.4.9 **public static final int RT_authorizationPolicy = 36**

resource type for authorizationPolicy

153.11.4.10 **public static final int RT_backgroundDataTransfer = 49**

resource type for backgroundDataTransfer

153.11.4.11 **public static final int RT_container = 3**

resource type for container

153.11.4.12 **public static final int RT_containerAnnc = 10003**

resource type for containerAnnc

153.11.4.13 **public static final int RT_contentInstance = 4**

resource type for contentInstance

153.11.4.14 **public static final int RT_contentInstanceAnnc = 10004**

resource type for contentInstanceAnnc

153.11.4.15 **public static final int RT_crossResourceSubscription = 48**

resource type for crossResourceSubscription

153.11.4.16 **public static final int RT_CSEBase = 5**

resource type for CSEBase

153.11.4.17 **public static final int RT_delivery = 6**

resource type for delivery

153.11.4.18 **public static final int RT_dynamicAuthorizationConsultation = 34**

resource type for dynamicAuthorizationConsultation

153.11.4.19 **public static final int RT_dynamicAuthorizationConsultationAnnc = 10034**

resource type for dynamicAuthorizationConsultationAnnc

153.11.4.20 **public static final int RT_eventConfig = 7**

resource type for eventConfig

153.11.4.21 **public static final int RT_execInstance = 8**

resource type for execInstance

153.11.4.22 **public static final int RT_flexContainer = 28**

resource type for flexContainer

153.11.4.23 **public static final int RT_flexContainerAnnc = 10028**

resource type for flexContainerAnnc

153.11.4.24 **public static final int RT_group = 9**

resource type for group

153.11.4.25 **public static final int RT_groupAnnc = 10009**

resource type for groupAnnc

153.11.4.26 **public static final int RT_localMulticastGroup = 45**

resource type for localMulticastGroup

153.11.4.27 **public static final int RT_locationPolicy = 10**

resource type for locationPolicy

153.11.4.28 **public static final int RT_locationPolicyAnnc = 10010**

resource type for locationPolicyAnnc

153.11.4.29 **public static final int RT_m2mServiceSubscriptionProfile = 11**

resource type for m2mServiceSubscriptionProfile

153.11.4.30 **public static final int RT_mgmtCmd = 12**

resource type for mgmtCmd

153.11.4.31 **public static final int RT_mgmtObj = 13**

resource type for mgmtObj

153.11.4.32 **public static final int RT_mgmtObjAnnc = 10013**

resource type for mgmtObjAnnc

153.11.4.33 **public static final int RT_multimediaSession = 46**

resource type for multimediaSession

153.11.4.34 **public static final int RT_multimediaSessionAnnc = 10046**

resource type for multimediaSessionAnnc

153.11.4.35 **public static final int RT_node = 14**

resource type for node

153.11.4.36 **public static final int RT_nodeAnnc = 10014**

resource type for nodeAnnc

153.11.4.37 **public static final int RT_notificationTargetMgmtPolicyRef = 25**

resource type for notificationTargetMgmtPolicyRef

153.11.4.38 **public static final int RT_notificationTargetPolicy = 26**

resource type for notificationTargetPolicy

153.11.4.39 **public static final int RT_ontology = 39**

resource type for ontology

153.11.4.40 **public static final int RT_ontologyAnnc = 10039**

resource type for ontologyAnnc

153.11.4.41 **public static final int RT_ontologyRepository = 38**

resource type for ontologyRepository

153.11.4.42 public static final int RT_ontologyRepositoryAnnc = 10038

resource type for ontologyRepositoryAnnc

153.11.4.43 public static final int RT_policyDeletionRules = 27

resource type for policyDeletionRules

153.11.4.44 public static final int RT_pollingChannel = 15

resource type for pollingChannel

153.11.4.45 public static final int RT_remoteCSE = 16

resource type for remoteCSE

153.11.4.46 public static final int RT_remoteCSEAnnc = 10016

resource type for remoteCSEAnnc

153.11.4.47 public static final int RT_request = 17

resource type for request

153.11.4.48 public static final int RT_role = 31

resource type for role

153.11.4.49 public static final int RT_schedule = 18

resource type for schedule

153.11.4.50 public static final int RT_scheduleAnnc = 10018

resource type for scheduleAnnc

153.11.4.51 public static final int RT_semanticDescriptor = 24

resource type for semanticDescriptor

153.11.4.52 public static final int RT_semanticDescriptorAnnc = 10024

resource type for semanticDescriptorAnnc

153.11.4.53 public static final int RT_semanticMashupInstance = 41

resource type for semanticMashupInstance

153.11.4.54 public static final int RT_semanticMashupInstanceAnnc = 10041

resource type for semanticMashupInstanceAnnc

153.11.4.55 public static final int RT_semanticMashupJobProfile = 40

resource type for semanticMashupJobProfile

153.11.4.56 public static final int RT_semanticMashupJobProfileAnnc = 10040

resource type for semanticMashupJobProfileAnnc

153.11.4.57 public static final int RT_semanticMashupResult = 42

resource type for semanticMashupResult

153.11.4.58 public static final int RT_semanticMashupResultAnnc = 10042

resource type for semanticMashupResultAnnc

153.11.4.59 public static final int RT_serviceSubscribedAppRule = 19

resource type for serviceSubscribedAppRule

153.11.4.60 public static final int RT_serviceSubscribedNode = 20

resource type for serviceSubscribedNode

153.11.4.61 public static final int RT_statsCollect = 21

resource type for statsCollect

153.11.4.62 public static final int RT_statsConfig = 22

resource type for

153.11.4.63 public static final int RT_subscription = 23

resource type for statsConfig

153.11.4.64 public static final int RT_timeSeries = 29

resource type for timeSeries

153.11.4.65 public static final int RT_timeSeriesAnnc = 10029

resource type for timeSeriesAnnc

153.11.4.66 public static final int RT_timeSeriesInstance = 30

resource type for timeSeriesInstance

153.11.4.67 public static final int RT_timeSeriesInstanceAnnc = 10030

resource type for timeSeriesInstanceAnnc

153.11.4.68 public static final int RT_token = 32

resource type for token

153.11.4.69 public static final int RT_transaction = 51

resource type for transaction

153.11.4.70 public static final int RT_transactionMgmt = 50

resource type for transactionMgmt

153.11.4.71 public static final int RT_triggerRequest = 47

resource type for triggerRequest

153.11.5 public class DasInfoDTO extends DTO

DTO expresses DasInfo. DAS is short for Dynamic Authorization Server.

See Also oneM2M TS-0004 6.3.5.45 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD dynAuthTokenReqInfo and dasInfo [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L1135-1147]

Concurrency Not Thread-safe

153.11.5.1 public GenericDTO dasRequest

Information to send to the Dynamic Authorization Server

See Also oneM2M XSD dynAuthDasRequest [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L1149-1198]

153.11.5.2 public String securedDasRequest

Secured Information to send to the Dynamic Authorization Server. JWS or JWE is assigned to this field.

153.11.5.3 public String uri

Dynamic Authorization Server URI

153.11.5.4 public DasInfoDTO()

153.11.6 public class FilterCriteriaDTO extends DTO

DTO expresses FilterCriteria. This data structure is used for searching resources.

See Also oneM2M TS-0004 6.3.5.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oenM2M TS-0004 7.3.3.17.17 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.6.1 public String applyRelativePath

Apply Relative Path

153.11.6.2 public List<AttributeDTO> attribute

Attribute

153.11.6.3 public List<AttributeDTO> childAttribute

Child Attribute

153.11.6.4 public List<String> childLabels

Child Labels

153.11.6.5 public List<Integer> childResourceType

Child Resource Type

153.11.6.6 public String contentFilterQuery

Content Filter Query

153.11.6.7 public Integer contentFilterSyntax

Content Filter Syntax

153.11.6.8 public List<String> contentType

Content Type

153.11.6.9 public String createdAfter

Created After

153.11.6.10 public String createdBefore

Created Before

153.11.6.11 **public String expireAfter**

Expire After

153.11.6.12 **public String expireBefore**

Expire Before

153.11.6.13 **public FilterCriteriaDTO.FilterOperation filterOperation**

Filter Operation

153.11.6.14 **public FilterCriteriaDTO.FilterUsage filterUsage**

Filter Usage

153.11.6.15 **public List<String> labels**

Labels

153.11.6.16 **public String labelsQuery**

Label Query

153.11.6.17 **public Integer level**

Level

153.11.6.18 **public Integer limit**

Limit number of Answers

153.11.6.19 **public String modifiedSince**

Modified Since

153.11.6.20 **public Integer offset**

Offset

153.11.6.21 **public List<AttributeDTO> parentAttribute**

Parent Attribute

153.11.6.22 **public List<String> parentLabels**

Parent Labels

153.11.6.23 **public List<Integer> parentResourceType**

Parent Resource Type

153.11.6.24 **public List<Integer> resourceType**

Resource Type

153.11.6.25 **public List<String> semanticsFilter**

Semantic Filter

153.11.6.26 **public Integer sizeAbove**

Size Above

153.11.6.27 **public Integer sizeBelow**

Size Below

153.11.6.28 **public Integer stateTagBigger**

State Tag Bigger

153.11.6.29 **public Integer stateTagSmaller**

State Tag Smaller

153.11.6.30 **public String unmodifiedSince**

Unmodified Since

153.11.6.31 **public FilterCriteriaDTO()**

153.11.7 enum FilterCriteriaDTO.FilterOperation

Enum FilterOperation

See Also oneM2M TS-0004 6.3.4.2.34 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.7.1 **AND**

Logical AND

153.11.7.2 **OR**

Logical OR

153.11.7.3 **public int getValue()**

□ get assigned value

Returns assigned integer value

153.11.7.4 **public static FilterCriteriaDTO.FilterOperation valueOf(String name)**

153.11.7.5 **public static FilterCriteriaDTO.FilterOperation[] values()**

153.11.8 enum FilterCriteriaDTO.FilterUsage

Enum FilterUsage

See Also oneM2M TS-0004 6.3.4.2.31 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.8.1 **DiscoveryCriteria**

Discovery Criteria

153.11.8.2 **ConditionalRetrival**

Conditional Retrieve

153.11.8.3 **IPEOndemandDiscovery**

IPE on Demand Discovery

153.11.8.4 **public int getValue()**

□ get assigned integer value

Returns assigned integer value

153.11.8.5 **public static FilterCriteriaDTO.FilterUsage valueOf(String name)**

153.11.8.6 public static FilterCriteriaDTO.FilterUsage[] values()

153.11.9 public class GenericDTO extends DTO

GenericDTO expresses miscellaneous data structures of oneM2M.

Concurrency Not Thread-safe

153.11.9.1 public Map<String, Object> element

Substructure of DTO. Type of the value part should be one of types allowed as OSGi DTO.

153.11.9.2 public GenericDTO()

153.11.10 public class IPEDiscoveryRequestDTO extends DTO

IPEDiscoveryRequestDTO is an element of NotificationEventDTO

See Also oneM2M TS-0004 6.3.5.13 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.10.1 public FilterCriteriaDTO filterCriteria

FilterCriteria

See Also oneM2M TS-0004 6.3.5.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.10.2 public String originator

originator

153.11.10.3 public IPEDiscoveryRequestDTO()

153.11.11 public class LocalTokenIdAssignmentDTO extends DTO

DTO expresses LocalTokenIdAssignment.

See Also oneM2M XSD dynAuthLocalTokenIdAssignments and localTokenIdAssignment [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L1112-1123]

Concurrency Not Thread-safe

153.11.11.1 public String localTokenID

local token ID

153.11.11.2 public String tokenID

token ID

153.11.11.3 public LocalTokenIdAssignmentDTO()

153.11.12 public class NotificationDTO extends DTO

DTO expresses Notification.

See Also oneM2M TS-0004 6.3.5.13 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.12.1 public Boolean aeReferenceIDChange

aeReferenceIDChange element

See Also oneM2M TS-0004 7.5.1.2.17 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.12.2 public Boolean aeRegistrationPointChange

AE Registration Point Change

See Also oneM2M TS-0004 7.5.1.2.16 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.12.3 public String creator

creator

153.11.12.4 public IPEDiscoveryRequestDTO ipeDiscoveryRequest

IPE Discovery Request.

153.11.12.5 public NotificationEventDTO notificationEvent

Notification Event

153.11.12.6 public String notificationForwardingURI

notification forwarding URI

153.11.12.7 public String notificationTarget

ID for notification target

153.11.12.8 public Boolean subscriptionDeletion

Flag showing subscription deletion This field is optional.

153.11.12.9 public String subscriptionReference

URI referring subscription resource.

153.11.12.10 public String trackingID1

tracking ID 1

See Also oneM2M TS-0004 7.5.1.2.16 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.5.1.2.17 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.12.11 public String trackingID2

tracking ID 1

See Also oneM2M TS-0004 7.5.1.2.16 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.5.1.2.17 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.12.12 public Boolean verificationRequest

Flag showing verification request. This field is optional.

153.11.12.13 public NotificationDTO()

153.11.13 public class NotificationEventDTO

DTO expresses NotificationEventDTO

This data structure is held in NotificationDTO.

See Also oneM2M TS-0004 6.3.5.13 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.13.1 public NotificationEventDTO.NotificationEventType notificationEventType

notificationEventType

See Also oneM2M TS-0004 6.3.4.2.19 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.13.2 public Map<String, Object> operationMonitor

operationMonitor

See Also oneM2M TS-0004 6.3.5.57 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.13.3 public Object representation

m2m:representation

See Also oneM2M TS-0004 6.3.5.62 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.13.4 public NotificationEventDTO()

153.11.14 enum NotificationEventDTO.NotificationEventType

NotificationEventType

See Also oneM2M TS-0004 6.3.4.2.19 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.14.1 update_of_resource

update_of_resouce. This is the default value.

153.11.14.2 delete_of_resource

delete_of_resource

153.11.14.3 create_of_direct_child_resource

create_of_direct_child_resource

153.11.14.4 delete_of_direct_child_resouce

create_of_direct_child_resouce

153.11.14.5 retrieve_of_container_resource_with_no_child_resource

retrieve_of_container_resource_with_no_child_resource

153.11.14.6 public int getValue()

□ Return notification type value.

Returns The notification type value.

153.11.14.7 public static NotificationEventDTO.NotificationEventType valueOf(String name)

153.11.14.8 public static NotificationEventDTO.NotificationEventType[] values()

153.11.15 public class PrimitiveContentDTO extends DTO

DTO expresses Primitive Content.

This Data structure is used as union. Only one field MUST have a value, the others MUST be null.

See Also oneM2M TS-0004 6.3.5.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.2.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD primitiveContent [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L596-602]

Concurrency Not Thread-safe

153.11.15.1 public List<NotificationDTO> aggregatedNotification

Aggregated Notification

153.11.15.2 public List<ResponsePrimitiveDTO> aggregatedResponse

Aggregated Response

153.11.15.3 public List<String> attributeList

Attribute List

153.11.15.4 public List<ChildResourceRefDTO> childResourceRefList

Child Resource RefList

153.11.15.5 public String debugInfo

Debug Info

153.11.15.6 public List<String> listOfURIs

List Of URIs

153.11.15.7 public NotificationDTO notification

Notification

153.11.15.8 public String queryResult

Query Result

153.11.15.9 public RequestPrimitiveDTO requestPrimitive

Request Primitive

153.11.15.10 public ResourceDTO resource

Resource

153.11.15.11 public ResourceWrapperDTO resourceWrapper

Resource Wrapper

153.11.15.12 **public ResponsePrimitiveDTO responsePrimitive**

Response Primitive

153.11.15.13 **public SecurityInfoDTO securityInfo**

Security Info

153.11.15.14 **public String uri**

URI

153.11.15.15 **public PrimitiveContentDTO()**

153.11.16 enum ReleaseVersion

enum expresses oneM2M specification version.

This information is introduced after Release 2.0 and oneM2M uses only R2A, R3_0 (as 2a and 3).

See Also oneM2M XSD releaseVersion [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L450-455]

153.11.16.1 **R1_0**

Release 1

153.11.16.2 **R1_1**

Release 1.1

153.11.16.3 **R2_0**

Release 2

153.11.16.4 **R2A**

Release 2A

153.11.16.5 **R3_0**

Release 3

153.11.16.6 **R4_0**

Release 4 (reserved for future)

153.11.16.7 **R5_0**

Release 5 (reserved for future)

153.11.16.8 **public static ReleaseVersion valueOf(String name)**

153.11.16.9 **public static ReleaseVersion[] values()**

153.11.17 public class RequestPrimitiveDTO extends DTO

DTO expresses Request Primitive.

See Also oneM2M TS-0004 6.4.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD requestPrimitive [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-requestPrimitive-v3_11_0.xsd]

Concurrency Not Thread-safe

153.11.17.1 public Boolean authorizationRelationshipIndicator

Authorization Relationship Indicator

153.11.17.2 public Boolean authorizationSignatureIndicator

Authorization Signature Indicator

153.11.17.3 public List<String> authorizationSignatures

Authorization Signatures

In oneM2M this parameter is expressed in m2m:signatureList.

See Also oneM2M TS-0004 6.3.4.2.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD signatureList [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L129-137]

153.11.17.4 public PrimitiveContentDTO content

Primitive Content

See Also oneM2M TS-0004 6.3.5.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.2.1.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.5 public Boolean deliveryAggregation

Delivery Aggregation

This parameter is related to CMDH(Communication Management and Delivery Handling) policy.

See Also oneM2M TS-0004 D.12 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.6 public RequestPrimitiveDTO.DesiredIdentifierResultType desiredIdentifierResultType

Desired Identifier Result Type

This parameter specifies identifier type in response, such as structured or unstructured. This parameter used to be Discovery Result Type in previous oneM2M release.

See Also oneM2M TS-0004 6.3.4.2.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.7 public Integer eventCategory

Event Category

allowed values are 2(Immediate), 3(BestEffort), 4(Latest), and 100-999 as user defined range.

See Also oneM2M TS-0004 6.3.3 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD eventCat [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L326], oneM2M XSD stdEventCats [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-enumerationTypes-v3_11_0.xsd#L208-221]

153.11.17.8 public FilterCriteriaDTO filterCriteria

Filter Criteria

See Also oneM2M TS-0004 6.3.5.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.9 public String from

From Parameter.

Originator of the request is stored.

See Also oneM2M TS-0004 6.3.4.2.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.10 public String groupRequestIdentifier

Group Request Identifier TODO: search doc.

153.11.17.11 public List<String> groupRequestTargetMembers

Group Request Target Members

153.11.17.12 public List<String> localTokenIDs

Local Token Identifiers

In oneM2M this parameter is expressed as list of xs:NCName.

153.11.17.13 public RequestPrimitiveDTO.Operation operation

Operation This field is mandatory.

See Also oneM2M TS-0004 6.3.4.2.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.14 public String operationExecutionTime

Operation Execution Time

153.11.17.15 public String originatingTimestamp

Originating Timestamp

153.11.17.16 public ReleaseVersion releaseVersionIndicator

Release Version

153.11.17.17 public String requestExpirationTimestamp

Request Expiration Timestamp

* This parameter is related to CMDH(Communication Management and Delivery Handling) policy.

See Also oneM2M TS-0004 D.12 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.18 public String requestIdentifier

Request Identifier

See Also oneM2M TS-0004 6.3.3 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.19 public Integer resourceType

Resource Type

See Also oneM2M TS-0004 6.3.4.2.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.20 public ResponseTypeInfoDTO responseType

Response Type Info

See Also oneM2M TS-0004 6.3.5.30 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.21 **public RequestPrimitiveDTO.ResultContent resultContent**

Result Content

See Also oneM2M TS-0004 6.3.4.2.7 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.22 **public String resultExpirationTimestamp**

Result Expiration Timestamp

This parameter is related to CMDH(Communication Management and Delivery Handling) policy.

See Also oneM2M TS-0004 D.12 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.23 **public String resultPersistence**

Result Persistence

This parameter is related to CMDH(Communication Management and Delivery Handling) policy.

See Also oneM2M TS-0004 D.12 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.24 **public List<String> roleIDs**

Role IDs

153.11.17.25 **public Boolean semanticQueryIndicator**

Semantic Query Indicator

See Also oneM2M TS-0004 7.3.3.19 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.17.26 **public String to**

To Parameter

153.11.17.27 **public List<String> tokenIDs**

Token Identifiers

In oneM2M this parameter is expressed as list of m2m:tokenID.

See Also oneM2M XSD signatureList [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L66-70]

153.11.17.28 **public Boolean tokenRequestIndicator**

Token Request Indicator

153.11.17.29 **public List<String> tokens**

Tokens

Each token is in m2m:dynAuthJWT

See Also oneM2M XSD signatureList [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L417-426]

153.11.17.30 **public String vendorInformation**

Vendor Information

Used for vendor specific information. No procedure is defined for the parameter.

153.11.17.31 **public RequestPrimitiveDTO()**

153.11.18 enum RequestPrimitiveDTO.DesiredIdentifierResultType

Enum for DesiredIdentifierResultType

See Also oneM2M TS-0004 6.3.4.2.8 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.18.1 structured

structured

153.11.18.2 unstructured

unstructured

153.11.18.3 public int getValue()

□ Return result type value.

Returns The result type value.

153.11.18.4 public static RequestPrimitiveDTO.DesiredIdentifierResultType valueOf(String name)

153.11.18.5 public static RequestPrimitiveDTO.DesiredIdentifierResultType[] values()

153.11.19 enum RequestPrimitiveDTO.Operation

enum type for Operation

See Also oneM2M XSD resultContent [https://git.onem2m.org/PRO/XSD/blob/master/v1_0_0/CDT-enumerationTypes-v1_0_0.xsd#L149-166]

153.11.19.1 Create

Create

153.11.19.2 Retrieve

Retrieve

153.11.19.3 Update

Update

153.11.19.4 Delete

Delete

153.11.19.5 Notify

Notify

153.11.19.6 public int getValue()

□ get assigned integer value

Returns assigned integer value

153.11.19.7 public static RequestPrimitiveDTO.Operation valueOf(String name)

153.11.19.8 public static RequestPrimitiveDTO.Operation[] values()

153.11.20 enum RequestPrimitiveDTO.ResultContent

enum type for Result Content

See Also oneM2M XSD resultContent [https://git.onem2m.org/PRO/XSD/blob/master/v1_0_0/CDT-enumerationTypes-v1_0_0.xsd#L183-205]

153.11.20.1 nothing

nothing

153.11.20.2 attributes

attributes

153.11.20.3 hierarchicalAddress

hierarchicalAddress

153.11.20.4 hierarchicalAddressAndAttributes

hierarchicalAddressAndAttributes

153.11.20.5 attributesAndChildResources

attributesAndChildResources

153.11.20.6 attributesAndChildResourceReferences

attributesAndChildResourceReferences

153.11.20.7 childResourceReferences

childResourceReferences

153.11.20.8 originalResource

originalResource

153.11.20.9 childResources

childResources

153.11.20.10 public int getValue()

□ get assigned integer value

Returns assigned integer value

153.11.20.11 public static RequestPrimitiveDTO.ResultContent valueOf(String name)

153.11.20.12 public static RequestPrimitiveDTO.ResultContent[] values()

153.11.21 public class ResourceDTO extends DTO

DTO expresses Resource.

Universal attributes are expressed in field of the class. Common attributes and other attributes are stored in attribute field.

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

Concurrency Not Thread-safe

153.11.21.1 public Map<String, Object> attribute

Non Universal Attribute. Value Part must be the types that are allowed for OSGi DTO. In case of value part can be expressed DTO in this package, the DTO must be used. In case of value part have sub-elements, GenericDTO must be used.

153.11.21.2 public String creationTime

Creation time

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

153.11.21.3 public String lastModifiedTime

last modified time

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

153.11.21.4 public String parentID

Parent ID Resource ID of parent resource.

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

153.11.21.5 public String resourceID

Resource ID

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

153.11.21.6 public String resourceName

Resource name

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf]

153.11.21.7 public Integer resourceType

Resource Type

See Also oneM2M TS-0001 9.6.1.3.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf], oenM2M TS-0004 6.3.4.2.1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.21.8 public ResourceDTO()

153.11.22 public class ResourceWrapperDTO extends DTO

DTO expresses ResourceWrapper.

See Also oneM2M TS-0004 6.3.5.25 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.22.1 public ResourceDTO resource

Resource

153.11.22.2 public String uri

Hierarchical URI of the resource

153.11.22.3 public ResourceWrapperDTO()

153.11.23 public class ResponsePrimitiveDTO extends DTO

DTO expresses Response Primitive.

See Also oneM2M TS-0004 6.4.2 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD responsePrimitive [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-responsePrimitive-v3_11_0.xsd]

Concurrency Not Thread-safe

153.11.23.1 public List<LocalTokenIdAssignmentDTO> assignedTokenIdentifiers

Assigned Token Identifiers

See Also oneM2M TS-0004 6.3.5.43 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.2 public Boolean AuthSignatureReqInfo

AuthSignatureReqInfo

153.11.23.3 public PrimitiveContentDTO content

Primitive Content

See Also oneM2M TS-0004 6.3.5.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.2.1.2 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.4 public Integer contentOffset

Content Offset

153.11.23.5 public ResponsePrimitiveDTO.ContentStatus contentStatus

Content Status

See Also oneM2M TS-0004 6.3.4.2.44 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.6 public Integer eventCategory

Event Category

allowed values are 2(Immediate), 3(BestEffort), 4(Latest), and 100-999 as user defined range.

See Also oneM2M TS-0004 6.3.3 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M XSD eventCat [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-commonTypes-v3_11_0.xsd#L326], oneM2M XSD stdEventCats [https://git.onem2m.org/PRO/XSD/blob/master/v3_11_0/CDT-enumerationTypes-v3_11_0.xsd#L208-221]

153.11.23.7 public String from

From Parameter

153.11.23.8 public String originatingTimestamp

Originating Timestamp To Parameter

See Also oneM2M TS-0004 Table 6.3.3-1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.9 public ReleaseVersion releaseVersionIndicator

Release Version Indicator

153.11.23.10 public String requestIdentifier

Request Identifier

See Also oneM2M TS-0004 6.3.3 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.11 public Integer responseStatusCode

Response Status Code

See Also oneM2M TS-0004 6.3.4.2.9 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.12 public String resultExpirationTimestamp

ResultExpiration Timestamp

See Also oneM2M TS-0004 Table 6.3.3-1 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.13 public String to

To Parameter

See Also oneM2M TS-0004 6.3.3 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.14 public List<DasInfoDTO> tokenReqInfo

Token Request Info

See Also oneM2M TS-0004 6.3.5.45 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.23.15 public String vendorInformation

Vendor Information

Used for vendor specific information. No procedure is defined for the parameter.

153.11.23.16 public ResponsePrimitiveDTO()

153.11.24 enum ResponsePrimitiveDTO.ContentStatus

Enum ContentStatus

See Also oneM2M TS-0004 6.3.4.2.44 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.24.1 PARTIAL_CONTENT

PARTIAL_CONTENT

153.11.24.2 FULL_CONTENT

FULL_CONTENT

153.11.24.3 public static ResponsePrimitiveDTO.ContentStatus valueOf(String name)

153.11.24.4 public static ResponsePrimitiveDTO.ContentStatus[] values()

153.11.25 public class ResponseTypeInfoDTO extends DTO

DTO expresses ResponseTypeInfo

See Also oneM2M TS-0004 6.3.5.30 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.25.1 public List<String> notificationURI

Notification URI

See Also oneM2M TS-0004 6.3.5.30 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf], oneM2M TS-0004 7.5.1.2.5 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.25.2 public ResponseTypeInfoDTO.ResponseType responseTypeValue

Response Type Value

See Also oneM2M TS-0004 6.3.4.2.6 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.25.3 public ResponseTypeInfoDTO()

153.11.26 enum ResponseTypeInfoDTO.ResponseType

enum ResponseType

See Also oneM2M TS-0004 6.3.4.2.6 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.26.1 nonBlockingRequestSynch

nonBlockingRequestSynch

153.11.26.2 nonBlockingRequestAsynch

nonBlockingRequestAsynch

153.11.26.3 blockingRequest

blockingRequest

153.11.26.4 flexBlocking

flexBlocking

153.11.26.5 public int getValue()

□ get assigned value

Returns assigned integer value.

153.11.26.6 public static ResponseTypeInfoDTO.ResponseType valueOf(String name)

153.11.26.7 public static ResponseTypeInfoDTO.ResponseType[] values()

153.11.27 public class SecurityInfoDTO extends DTO

DTO expresses Security Info.

This class is used as union. SecurityInfoType field indicates which type of content is stored.

See Also oenM2M TS-0004 6.3.5.48 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

Concurrency Not Thread-safe

153.11.27.1 public GenericDTO dasRequest

Das Request

153.11.27.2 public GenericDTO dasResponse

Das Response

153.11.27.3 public byte[] escertkeMessage

Escertke Message

153.11.27.4 public String esprimObject

Esprim Object

153.11.27.5 public GenericDTO esprimRandObject

Esprim Rand Object

153.11.27.6 public SecurityInfoDTO.SecurityInfoType securityInfoType

Security Info Type

See Also oenM2M TS-0004 6.3.4.2.35 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.27.7 public SecurityInfoDTO()

153.11.28 enum SecurityInfoDTO.SecurityInfoType

Enum SecurityInfoType

See Also oenM2M TS-0004 6.3.4.2.35 [http://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf]

153.11.28.1 DynamicAuthorizationRequest

DynamicAuthorizationRequest

153.11.28.2 DynamicAuthorizationResponse

DynamicAuthorizationResponse

153.11.28.3 ReceiverESPrimRandObjectRequest

ReceiverESPrimRandObjectRequest

153.11.28.4 ReceiverESPrimRandObjectResponse

ReceiverESPrimRandObjectResponse

153.11.28.5 ESPrimObject

ESPrimObject

153.11.28.6 ESCertKEMessage

ESCertKEMessage

153.11.28.7 DynamicAuthorizationRelationshipMappingRequest

DynamicAuthorizationRelationshipMappingRequest

153.11.28.8 DynamicAuthorizationRelationshipMappingResponse

DynamicAuthorizationRelationshipMappingResponse

153.11.28.9 public int getValue()

□ Get assigned value.

Returns assigned value

153.11.28.10 public static SecurityInfoDTO.SecurityInfoType valueOf(String name)

153.11.28.11 public static SecurityInfoDTO.SecurityInfoType[] values()

153.12 References

[1] *oneM2M: TS-0001 Functional Architecture V3.15.1*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0001-Functional_Architecture-V3_15_1.pdf

[2] *oneM2M: TS-0004 Service Layer Core Protocol V3.11.2*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0004_Service_Layer_Core_Protocol_V3_11_2.pdf

[3] *oneM2M: TS-0008 CoAP Protocol Binding V3.3.1*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0008-CoAP_Protocol_Binding-V3_3_1cl.pdf

[4] *oneM2M: TS-0009 HTTP Protocol Binding V3.2.1*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0009-HTTP_Protocol_Binding-V3_2_1cl.pdf

[5] *oneM2M: TS-0010 MQTT Protocol Binding V3.0.1*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0010-MQTT_protocol_binding-V3_0_1.pdf

[6] *oneM2M: TS-0020 WebSocket Protocol Binding V3.0.1*
https://www.onem2m.org/images/files/deliverables/Release3/TS-0020-WebSocket_Protocol_Binding-V3_0_1.pdf

[7] *oneM2M-schemas*
https://git.onem2m.org/PRO/XSD

154 Residential Device Management Tree Specification

Version 1.0

154.1 Introduction

The chapter defines the Device Management Tree (DMT) for residential applications called the *Residential Management Tree* (RMT). This RMT is based on the *Dmt Admin Service Specification* on page 365. The RMT allows remote managers to manage the residential device through an abstract tree. As this tree is an abstract representation, different management protocols can use the same underlying management components, the Dmt Admin Plugins, in the OSGi framework.

This chapter requires full understanding of the concepts in the *Dmt Admin Service Specification* on page 365 and uses its terminology.

154.1.1 Essentials

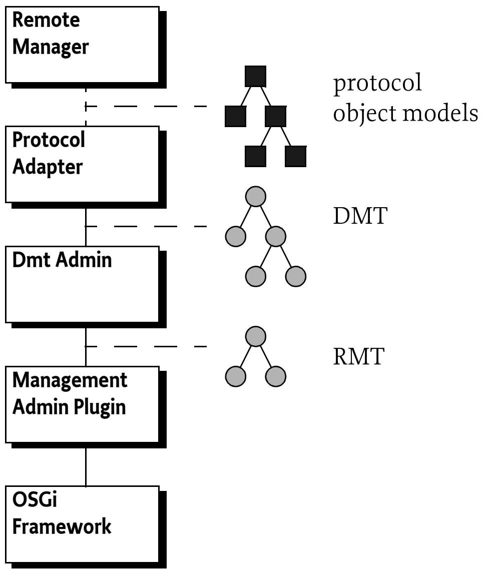
The following essentials are associated with the Residential Management Tree specification:

- *Complete* - The RMT must cover all functionality to completely manage an OSGi Framework as defined by *OSGi Core Release 8*.
- *Performance* - The RMT runs on devices with limited resources.
- *Searchable* - Provide an efficient way to search the RMT remotely.
- *Services* - Provide efficient access to standardized services like the Log Service.

154.1.2 Entities

- *Remote Manager* - The entity that remotely controls an OSGi Framework.
- *Management Agent* - An entity running on the device that is responsible for the management of the local OSGi Framework. It usually acts as a proxy for a Remote Manager.
- *Protocol Adapter* - Communicates with a Remote Manager and translates the protocol instructions to instructions to a local Management Agent.
- *DMT* - The Device Management Tree. This is the general structure available through the Dmt Admin service.
- *RMT* - The Residential Management Tree. This is the part of the DMT that is involved with residential management.

Figure 154.1 Device Management Architecture



154.2 The Residential Management Tree

The *OSGi* node is the root node for OSGi specific information. This OSGi node can be placed anywhere in the Device Management Tree and acts as parent to all the top level nodes in this specification. Therefore, in this specification the parent node of, for example, the Framework node is referred to as $, which effectively represents the OSGi node. The description of the nodes are using the types defined in *OSGi Object Modeling* on page 407.

The value of $ for a specific system can be defined with the following Framework property:

```
org.osgi.dmt.residential
```

For this specifications, the RMT Consists of the following top level nodes:

- Framework - Managing the local Framework
- Filter - Searching nodes in the DMT
- Log - Access to the log

154.3 Managing Bundles

The Framework node provides a remote management model for managing the life cycle of bundles and inspecting the Framework's state.

To change the state, for example install a new bundle, requires an atomic session on at least the Framework node. The model is constructed to reflect the requested state. When the session is committed, the underlying Plugin must effectuate these requested states into the real state.

For example, to install a bundle it is first necessary to create a new Bundle child node. The Bundle node is a MAP node, the name of the child node is the location of the bundle as given in the installBundle(location,input stream) method and returned from the getLocation() method.

This location should not be treated as the actual URL of the bundle, the location is better intended to be used a management name for the bundle as the remote manager can choose it. It is normally best to make this name a reverse domain name, for example com.acme.admin. The name "System Bundle" is a reserved name for the system bundle. The Framework management plugin must therefore not treat the location as a URL.

Creating the child node has no effect as long as the session is not committed. This new Bundle node automatically gets the members defined in the Bundle type.

The URL node should be set to the download URL, the URL used to download the JAR file from. The URL node is used as the download URL for an install operation (after the node is created newly) or the update location when the node is changed after the bundle had been installed in a previous session. Creating a new Bundle node without setting the URL must generate an error when the session is committed.

To start this newly installed bundle, the manager can set the RequestedState to ACTIVE. If this bundle needs to be started when the framework is restarted, then the AutoStart node can be set to true. If there bundles to be uninstalled then their RequestedState node must be set to UNINSTALLED as it is not possible to delete a Bundle node. The RequestedState must be applied after the bundle has been installed or updated. An uninstalled bundle will be automatically removed from the RMT.

The RequestedState node is really the requested state, depending on start levels and other existing conditions the bundle can either follow the requested state or have another state if, for example, its start level is not met. The RequestedState must be stored persistently between invocations, its initial value is INSTALLED.

The manager can create any number of new Bundle nodes to install a number of bundles at the same time during commit. It can also change the life cycle of existing bundles. None of these changes must have any effect until the session is committed.

If the session is finally committed, the Plugin must compare the state in the Dmt Admin tree with the actual state and update the framework accordingly. The order in which the operations occur is up to the implementation except for framework operations, they must always occur last. After bundles have been installed, uninstalled, or updated, the Plugin must refresh all the packages to ensure that the remote management system sees a consistent state.

Downloading the bundles from a remote system can take substantial time. As the commit is used synchronously, it is sometimes advisable to download the bundles to the device before they are installed.

If any error occurs, any changes that were made since the beginning of the last transaction point must be rolled back. An error should be reported. The remote manager therefore gets an atomic behavior, either all changes succeed or all fail. A manager should also be aware that if its own bundle, or any of its dependencies, is updated it will be stopped and will not be able to properly report the outcome to the management system, either a failure or success.

154.3.1 Bundle Life Cycle Example

For example, the following code installs my_bundle, updates up_bundle, and uninstalls old_bundle:

```
String $ = ... // get the OSGi node
DmtSession session = admin.getSession($ + "/Framework",
                DmtSession.LOCK_TYPE_ATOMIC);
try {
  session.createInteriorNode("Bundle/my_bundle");
  session.setNodeValue("Bundle/my_bundle/URL", new DmtData(
    "http://www.example.com/bundles/my_bundle.jar"));
  session.setNodeValue("Bundle/my_bundle/AutoStart",
    DmtData.TRUE_VALUE);
  session.setNodeValue("Bundle/my_bundle/RequestedState",
    new DmtData("ACTIVE"));
  session.setNodeValue("Bundle/up_bundle/URL", new DmtData(
    "http://www.example.com/bundles/up_bundle-2.jar"));

  session.setNodeValue("Bundle/old_bundle/RequestedState",
    new DmtData("UNINSTALLED"));
  try {
    session.commit();
  } catch (Exception e) {
     // failure ...
     log....
  }
} catch (Exception e) {
  session.rollback();
  log...
}
```

154.3.2 Framework Restart

There are no special operations for managing the life cycle of the Framework, these operations are done on the System Bundle, or bundle 0. The framework can be stopped or restarted:

- *Restart* - Restarting is an update, requiring the URL to be set to a new URL. This must shutdown the framework after the commit has succeeded.
- *Stopping* - Stopping is setting the RequestedState to INSTALLED

If the URL node has changed, the RequestedState will be ignored and the framework must only be restarted.

Sessions that modify nodes inside the Framework sub-tree must always be atomic and opened on the Framework node. The Data Plugin managing the Framework node is only required to handle a single simultaneous atomic session for its whole sub-tree.

For example, the following code restarts the framework after the commit has succeeded.

```
DmtSession session = admin.getSession($ +"/Framework",
                DmtSession.LOCK_TYPE_ATOMIC);
session.setNodeValue("Bundle/System Bundle/URL",
  new DmtData(""));
session.commit();
```

154.3.3 Access to Wiring

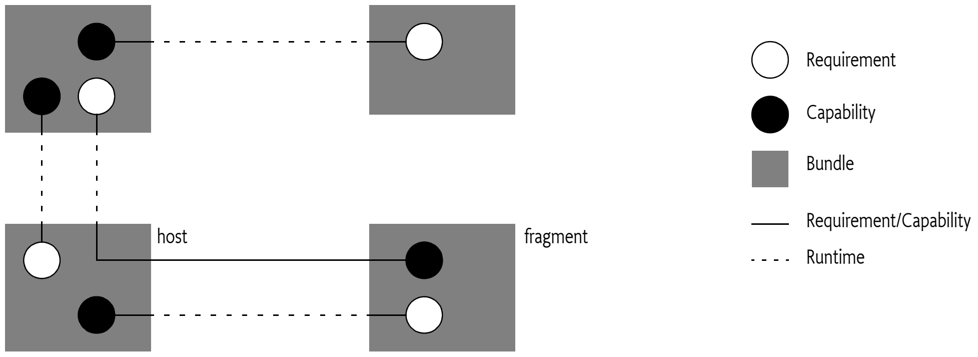
During runtime a bundle is wired to several different entities, other bundles, fragments, packages, and services. The framework defines a general Requirement-Capability model and this model is reflected in the Wiring API in *OSGi Core Release 8*. The Requirement-Capability model maps to a very generic way of describing wires between requirers and providers that is applicable to all of the OSGi constructs.

The Core defines namespaces for:

- osgi.wiring.bundle - The namespace for the Require-Bundle header. It wires the bundle with the Require-Bundle header to the bundle with the required Bundle-SymbolicName and Bundle-Version header.
- osgi.wiring.host - The namespace for the Fragment-Host header. It wires from bundle with the Fragment-Host header to the bundle with the required Bundle-SymbolicName and Bundle-Version header.
- osgi.wiring.package - The namespace for the Import/Export-Package header. It wires from bundle with the Import-Package header to the bundle with the Export-Package header.

In the Core API, the wiring is based on the Bundle revisions. However, this specification requires that all bundles are refreshed after a management operation to ensure a consistent wiring state. The management model therefore ignores the Bundle Revision and instead provides wiring only for bundles since the manager is unable to see different revision of a bundle anyway. The general Requirement-Capability model is depicted in Figure 154.2.

Figure 154.2 Requirements and Capabilities and their Wiring



The core does not specify a namespace for services. However, services can also be modeled with requirements capabilities. The registrar is the provider and the service properties are the capability. The getter is the requirer, its filter is the requirement. This specification therefore also defines a namespaces for services:

```
osgi.wiring.rmt.service
```

This namespace is defined in *osgi.wiring.rmt.service Namespace* on page 1310.

To access the wiring, each Bundle node has a Wires node. This is a MAP of LIST of Wire. The key of the MAP node is the name of the namespace, that is, the wires are organized by namespace. This provides convenient access to all wires of a given namespace. The value of the MAP node is a LIST node, providing sequential access to the actual wires.

A Wire node provides the following information:

- Namespace - The namespace of the wire
- Requirement - The requirement that cause the wire
- Capability - The capability that satisfied the wire
- Requirer - The location of the bundle that required the wire
- Provider - The location of the bundle that satisfied the requirement

154.3.4 Wiring Example

The following example code demonstrates how the wires can be printed out:

```
String prefix ="Bundle/my_bundle/Wires/osgi.wiring.package";
String [] wires = session.getChildNodeNames(prefix);
for ( String wire : wires ) {
  String name = session.getNodeValue(prefix + "/"
     + wire + "/Capability/Attribute/osgi.wiring.package").getString();
  String provider = session.getNodeValue(prefix + "/"
     + wire + "/Provider" ).getString();
  String requirer = session.getNodeValue(prefix + "/"
     + wire + "/Requirer" ).getString();
  System.out.printf("%-20s %-30s %s\n", name, provider, requirer);
}
```

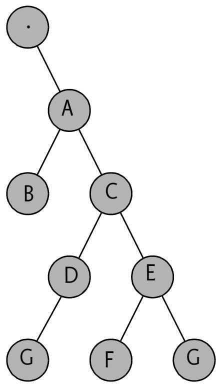
154.4 Filtering

Frequently it is necessary to search through the tree of nodes for nodes matching specific criteria. Having to use Java to do this filtering can become cumbersome and impossible if the searching has to happen remotely. For that reason, the RMT contains a Filter node. This node allows a manager to specify a Target and a Filter. The Target is an absolute URI that defines a set of nodes that the Filter Plugin must search. This set is defined by allowing wildcards in the target. A single asterisk ('*' \u002A) matches a single level, the minus sign ('-' \u002C) specifies any number of levels and must not be used at the end of the URI. This implies that there is always a *final node*. The reason that a minus sign must not be last is that the final node's type would be undefined, any node on any sub-level would match.

The Target node must be specified as an absolute URI that must always end in a solidus ('/' \u002F) to signify that it represents a path to an interior node. The URI is absolute because the Filter is specified in a persistent node. It is possible to open a session, create the filter specification, close the session, and then open a new session, and use the earlier specified Target. As the two involved session do not have to have the same session, the base could differ, making it hard to use relative addressing. However, the result is always unique to a session. It is therefore possible to use relative URIs in the read out of the result.

For example, the tree in Figure 154.3 defines a sub-tree.

Figure 154.3 *Example Sub-Tree*



The following table shows a number of example targets on the previous sub-tree and their resulting final nodes, assuming the result is read in a session open on ./A.

Table 154.1 *Example Target and results on a session opened on ./A*

Target	Final nodes
./A/*/	B, C
./A/*/E/*/	C/E/F, C/E/G
./A/-/G/	C/D/G, C/E/G
./A/*/*/*/	C/D/G, C/E/F, C/E/G
./A/-/*/	This is an error,./A/-/*/ is the same as ./A/-/, which is not allowed.
./A/*/*/	C/D, C/E

The Filter specifies a standard OSGi Filter expression that is applied to the final nodes. If no filter is specified then all final nodes match. However, when there is a filter specified it is applied against the final node and only the final nodes that are matching the filter as included in the result.

A node is matched against a filter by using some of its children as properties. The properties of a node are defined by:

- Primitive child nodes, or
- LIST nodes that have primitive as child nodes. Such nodes must be treated as multi-valued properties.

The matching rules in the filter must follow the standard OSGi Filter rules. If the filter matches such a node then it must be available as a session relative URI in the ResultUriList node. The relative URIs are listed in the ResultUriList.

The result nodes must only include nodes that satisfy the following conditions:

- The node must match the Target node's URI specification
- The node must be visible in the current session
- The node must not reside in the Filter sub-tree
- The node must be an interior node
- The caller must have access to the node
- It must be possible to get all the values of the child nodes that are necessary for filter matching
- The node must match the filter if a filter is specified

The result is also available as a sub-tree under the Result node and can be traversed as sub-tree in Result. This tree contains all the result nodes and their sub-tree. The results under the Result node are a snapshot and cannot be modified, they are read only. This result can be removed after the session is closed.

154.4.1 Example

For example, the following code prints out the location of active bundles:

```
session.createInteriorNode("Filter/mq-1");
session.setNodeValue("Filter/mq-1/Target",
  new DmtData($+"/Framework/Bundle/*/"));
session.setNodeValue("Filter/mq-1/Filter", new DmtData("(AutoStart=true)"));

String[] autostarted = session.getChildNodeNames(
   "Filter/mq-1/Result/Framework/Bundle");
System.out.println("Auto started bundles");
for ( String location : autostarted)
  System.out.println(location);

session.deleteNode("Filter/mq-1");
```

154.5 Log Access

The Log node provides access to the Log Service, the node contains a LIST of LogEntry nodes. The length of this list is implementation dependent. The list is sorted in most recent first order. This allows a manager to retrieve the latest logs. For example, the following code print out the latest 100 log entries:

```
DataSession session = admin.getSession($+"/Log/LogEntries");
```

```
try {
  for ( int i =0; i<100; i++ ) {
    Date date = session.getNodeValue( i+"/Time").getDateTime();
    String message = session.getNodeValue( i+"/Message").getString();
    System.out.println( date + " " + message );
  }
} finally {
  session.close();
}
```

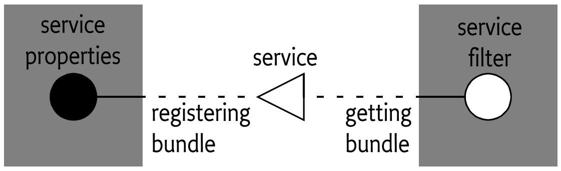
154.6 osgi.wiring.rmt.service Namespace

This section defines a namespace for the Requirement-Capability model to maintain services through the standard wiring API. A service is a capability, the Capability attributes are the service properties. The bundle that gets the service has a requirement on that service.

The filter of the service requirement is not the original filter since this is not possible to obtain reliably. Instead the filter must assert of the service.id, for example: (service.id=123).

The resulting filter is specified as the filter: directive on the Requirement. This is depicted in Figure 154.4.

Figure 154.4 Requirements and Capabilities and their Wiring



The osgi.wiring.rmt.service attributes are defined in the following table.

Table 154.2 osgi.wiring.rmt.service namespace

Attribute Name	Type	Syntax	Description
osgi.wiring.rmt.service	String	service.id	The service id.
objectClass	String[]	fqn	Fully qualified name of the types under which this service is listed
*	*	*	Any service property

154.7 Tree Summary

```
$                                          _G__  NODE                1   P
```

154.7.1 Filter

```
                                                            org.osgi/1.0/MAP
  Filter                                   _G__  MAP               0,1   P
    [string]                               AG_D  NODE             0..*   D
      Filter                               _GR_  string              1   A
      InstanceId                           _G__  integer             1   A
      Limit                                _GR_  integer             1   A
      Result                              _G__  Node                   1   A

                                                           org.osgi/1.0/LIST
      ResultUriList                       _G__  LIST                   1   A
        [list]                            _G__  string             0..*    D
      Target                              _GR_  string                 1   A
```

154.7.2 Framework

```
  Framework                               _G__  NODE                   1   P

                                                            org.osgi/1.0/MAP
    Bundle                                _G__  MAP                    1   A
      [string]                            AG__  NODE               0..*    D
        AutoStart                         _GR_  boolean                1   A
        BundleId                          _G__  long                 0,1   A

                                                           org.osgi/1.0/LIST
        BundleType                        _G__  LIST                 0,1   A
          [list]                          _G__  string             0..*    D

                                                           org.osgi/1.0/LIST
        Entries                           _G__  LIST                 0,1   A
          [list]                          _G__  NODE               0..*    D
            Content                       _G__  binary                 1   A
            InstanceId                    _G__  integer                1   A
            Path                          _G__  string                 1   A
        FaultMessage                      _G__  string               0,1   A
        FaultType                         _G__  integer              0,1   A

                                                            org.osgi/1.0/MAP
        Headers                           _G__  MAP                  0,1   A
          [string]                        _G__  string             0..*    D
        InstanceId                        _G__  integer                1   A
        LastModified                      _G__  date_time            0,1   A
        Location                          _G__  string                 1   A
        RequestedState                    _GR_  string                 1   A

                                                           org.osgi/1.0/LIST
        Signers                           _G__  LIST                 0,1   A
          [list]                          _G__  NODE               0..*    D

                                                           org.osgi/1.0/LIST
            CertificateChain              _G__  LIST                   1   A
              [list]                      _G__  string             0..*    D
            InstanceId                    _G__  integer                1   A
            IsTrusted                     _G__  boolean                1   A
        StartLevel                        _GR_  integer                1   A
        State                             _G__  string               0,1   A
        SymbolicName                      _G__  string               0,1   A
        URL                               _GR_  string                 1   A
        Version                           _G__  string               0,1   A

                                                            org.osgi/1.0/MAP
        Wires                             _G__  MAP                  0,1   A
```

```
                                                                  org.osgi/1.0/LIST
      [string]                        _G__  LIST                          0..*   D
        [list]                        _G__  NODE                          0..*   D
          Capability                  _G__  NODE                             1   A

                                                                   org.osgi/1.0/MAP
            Attribute                 _G__  MAP                              1   A
              [string]                _G__  string                        0..*   D

                                                                   org.osgi/1.0/MAP
            Directive                 _G__  MAP                              1   A
              [string]                _G__  string                        0..*   D
          InstanceId                  _G__  integer                          1   A
          Namespace                   _G__  string                           1   A
          Provider                    _G__  string                           1   A
          Requirement                 _G__  NODE                             1   A

                                                                   org.osgi/1.0/MAP
            Attribute                 _G__  MAP                              1   A
              [string]                _G__  string                        0..*   D

                                                                   org.osgi/1.0/MAP
            Directive                 _G__  MAP                              1   A
              [string]                _G__  string                        0..*   D
            Filter                    _G__  string                           1   A
          Requirer                    _G__  string                           1   A
  InitialBundleStartLevel             _GR_  integer                          1   A

                                                                   org.osgi/1.0/MAP
  Property                            _G__  MAP                              1   A
    [string]                          _G__  string                        0..*   D
  StartLevel                          _GR_  integer                          1   A
```

154.7.3 Log

```
Log                                   _G__  NODE                           0,1   P

                                                                  org.osgi/1.0/LIST
  LogEntries                          _G__  LIST                             1   A
    [list]                            _G__  NODE                          0..*   D
      Bundle                          _G__  string                           1   A
      Exception                       _G__  string                         0,1   A
      Level                           _G__  integer                          1   A
      Message                         _G__  string                           1   A
      Time                            _G__  date_time                        1   A
```

154.8 org.osgi.dmt.residential

154.8.1 $

The $ describes the root node for OSGi Residential Management. The path to this node is defined in the system property: org.osgi.dmt.residential .

Table 154.3 Sub-tree Description for $

Name	Act	Type	Card.	S	Description
Framework	Get	Framework	1	P	The Framework node used to manage the local framework.
Filter	Get	MAP	0,1	P	The Filter node searches the nodes in a tree that correspond to a target URI and an optional filter expression. A new Filter is created by adding a node to the Filter node. The name of the node is chosen by the remote manager. If multiple managers are active they must agree on a scheme to avoid conflicts or an atomic sessions must be used to claim exclusiveness. Filter nodes are persistent but an implementation can remove the node after a suitable timeout that should at least be 1 hour. If this functionality is not supported on this device then the node is not present.
[String]	Add Del Get	Filter	0..*	D	
Log	Get	Log	0,1	P	Access to the optional Log. If this functionality is not supported on this device then the node is not present.

154.8.2 Bundle

The management node for a Bundle. It provides access to the life cycle control of the bundle as well to its metadata, resources, and wiring.

To install a new bundle an instance of this node must be created. Since many of the sub-nodes are not yet valid as the information from the bundle is not yet available. These nodes are marked to be optional and will only exists after the bundle has been really installed.

154.8.2.1 FRAGMENT = "FRAGMENT"

The type returned for a fragment bundle.

154.8.2.2 INSTALLED = "INSTALLED"

The Bundle INSTALLED state.

154.8.2.3 RESOLVED = "RESOLVED"

The Bundle RESOLVED state.

154.8.2.4 STARTING = "STARTING"

The Bundle STARTING state.

154.8.2.5 ACTIVE = "ACTIVE"

The Bundle ACTIVE state.

154.8.2.6 STOPPING = "STOPPING"

The Bundle STOPPING state.

154.8.2.7 UNINSTALLED = "UNINSTALLED"

The Bundle UNINSTALLED state.

Table 154.4 Sub-tree Description for Bundle

Name	Act	Type	Card.	S	Description
URL	Get Set	string	1	A	The URL to download the archive from for this bundle. By default this is the empty string. In an atomic session this URL can be replaced to a new URL, which will trigger an update of this bundle during commit. If this value is set it must point to a valid JAR from which a URL can be downloaded, unless it is the system bundle. If it is the empty string no action must be taken except when it is the system bundle. If the URL of Bundle 0 (The system bundle) is replaced to any value, including the empty string, then the framework will restart. If both a the URL node has been set the bundle must be updated before any of the other aspects are handled like RequestedState and StartLevel.
AutoStart	Get Set	boolean	1	A	Indicates if this Bundle must be started when the Framework is started. If the AutoStart node is true then this bundle is started when the framework is started and its StartLevel is met. If the AutoStart node is set to true and the bundle is not started then it will automatically be started if the start level permits it. If the AutoStart node is set to false then the bundle must not be stopped immediately. If the AutoStart value of the System Bundle is changed then the operation must be ignored. The default value for this node is true
FaultType	Get	integer	0,1	A	The BundleException type associated with a failure on this bundle, -1 if no fault is associated with this bundle. If there was no Bundle Exception associated with the failure the code must be 0 (UNSPECIFIED). The FaultMessage provides a human readable message. Only present after the bundle is installed.
FaultMessage	Get	string	0,1	A	A human readable message detailing an error situation or an empty string if no fault is associated with this bundle. Only present after the bundle is installed.
BundleId	Get	long	0,1	A	The Bundle Id as defined by the getBundleId() method. If there is no installed Bundle yet, then this node is not present.

Name	Act	Type	Card.	S	Description
SymbolicName	Get	string	0,1	A	The Bundle Symbolic Name as defined by the Bundle getSymbolicName() method. If this result is null then the value of this node must be the empty string. If there is no installed Bundle yet, then this node is not present.
Version	Get	string	0,1	A	The Bundle's version as defined by the Bundle getVersion() method. If there is no installed Bundle yet, then this node is not present.
BundleType	Get	LIST	0,1	A	A list of the types of the bundle. Currently only a single type is provided: • FRAGMENT If there is no installed Bundle yet, then this node is not present.
[list]	Get	string	0..*	D	
Headers	Get	MAP	0,1	A	The Bundle getHeaders() method.
[String]	Get	string	0..*	D	If there is no installed Bundle yet, then this node is not present.
Location	Get	string	1	A	The Bundle's Location as defined by the Bundle getLocation() method. The location is specified by the management agent when the bundle is installed. This location should be a unique name for a bundle chosen by the management system. The Bundle Location is immutable for the Bundle's life (it is not changed when the Bundle is updated). The Bundle Location is also part of the URI to this node.
State	Get	string	0,1	A	Return the state of the current Bundle. The values can be: • INSTALLED • RESOLVED • STARTING • ACTIVE • STOPPING If there is no installed Bundle yet, then this node is not present. The default value is UNINSTALLED after creation.

Name	Act	Type	Card.	S	Description
RequestedState	Get Set	string	1	A	Is the requested state the manager wants the bundle to be in. Can be: • INSTALLED - Ensure the bundle is stopped and refreshed. • RESOLVED - Ensure the bundle is resolved. • ACTIVE - Ensure the bundle is started. • UNINSTALLED - Uninstall the bundle. The Requested State is a request. The management agent must attempt to achieve the desired state but there is a no guarantee that this state is achievable. For example,a Framework can resolve a bundle at any time or the active start level can prevent a bundle from running. Any errors must be reported on FaultType and FaultMessage. If the AutoStart node is `true` then the bundle must be persistently started, otherwise it must be transiently started. If the StartLevel is not met then the commit must fail if AutoStart is `false` as a Bundle cannot be transiently started when the start level is not met. If both a the URL node has been set as well as the RequestedState node then this must result in an update after which the bundle should go to the RequestedState. The RequestedState must be stored persistently so that it contains the last requested state. The initial value of the RequestedState must be INSTALLED.
StartLevel	Get Set	integer	1	A	The Bundle's current Start Level as defined by the BundleStartLevel adapt interface `getStartLevel()` method. Changing the StartLevel can change the Bundle State as a bundle can become eligible for starting or stopping. If the URL node is set then a bundle must be updated before the start level is set,
LastModified	Get	date_time	0,1	A	The Last Modified time of this bundle as defined by the Bundle `getlastModified()` method. If there is no installed Bundle yet then this node is not present.

Name	Act	Type	Card.	S	Description
Wires	Get	MAP	0,1	A	A MAP of name space -> to Wire. A Wire is a relation between to bundles where the type of the relation is defined by the name space. For example, osgi.wiring.package name space defines the exporting and importing of packages. Standard osgi name spaces are: • osgi.wiring.bundle • osgi.wiring.package • osgi.wiring.host As the Core specification allows custom name spaces this list can be more extensive. This specification adds one additional name space to reflect the services, this is the osgi.wiring.rmt.service name space. This name space will have a wire for each time a registered service by this Bundle was gotten for the first time by a bundle. A capability in the service name space holds all the registered service properties. The requirement has no attributes and a single filter directive that matches the service id property. If there is no installed Bundle yet then this node is not present.
[String]	Get	LIST	0..*	D	
[list]	Get	Wire	0..*	D	
Signers	Get	LIST	0,1	A	Return all signers of the bundle. See the Bundle getSignerCertificates() method with the SIGNERS_ALL parameter. If there is no installed Bundle yet then this node is not present.
[list]	Get	Certificate	0..*	D	
Entries	Get	LIST	0,1	A	An optional node providing access to the entries in the Bundle's JAR. This list must be created from the Bundle getEntryPaths() method called with an empty String. For each found entry, an Entry object must be made available. If there is no installed Bundle yet then this node is not present.
[list]	Get	Entry	0..*	D	
InstanceId	Get	integer	1	A	Instance Id used by foreign protocol adapters as a unique integer key not equal to 0. The instance id for a bundle must be (Bundle Id % 2^32) + 1.

154.8.3 Bundle.Certificate

Place holder for the Signers DN names.

Table 154.5 Sub-tree Description for Bundle.Certificate

Name	Act	Type	Card.	S	Description
IsTrusted	Get	boolean	1	A	Return if this Certificate is trusted.
CertificateChain	Get	LIST	1	A	A list of signer DNs of the certificates in the chain.
[list]	Get	string	0..*	D	

Name	Act	Type	Card.	S	Description
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

154.8.4 Bundle.Entry

An Entry describes an entry in the Bundle, it combines the path of an entry with the content. Only entries that have content will be returned, that is, empty directories in the Bundle's archive are not returned.

Table 154.6 Sub-tree Description for Bundle.Entry

Name	Act	Type	Card.	S	Description
Path	Get	string	1	A	The path in the Bundle archive to the entry.
Content	Get	binary	1	A	The binary content of the entry.
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

154.8.5 Filter

A Filter node can find the nodes in a given sub-tree that correspond to a given filter expression. This Filter node is a generic mechanism to select a part of the sub-tree (*except* itself).

Searching is done by treating an interior node as a map where its leaf nodes are attributes for a filter expression. That is, an interior node matches when a filter matches on its children. The matching nodes' URIs are gathered under a ResultUriList node and as a virtual sub-tree under the Result node.

The Filter node can specify the Target node. The Target is an absolute URI ending in a slash, potentially with wild cards. Only nodes that match the target node are included in the result.

There are two different wild cards:

- *Asterisk* ('*' \u002A) - Specifies a wild card for one interior node name only. That is A/*/ matches an interior nodes A/B, A/C, but not A/X/Y. The asterisk wild card can be used anywhere in the URI like A/*/C. Partial matches are not supported, that is a URI like A/xyz* is invalid.
- *Minus sign* ('-' \u002D) - Specifies a wildcard for any number of descendant nodes. This is A/-/X/ matches A/B/X, A/C/X, but also A/X. Partial matches are not supported, that is a URI like A/xyz- is not supported. The - wild card must not be used at the last segment of a URI

The Target node selects a set of nodes N that can be viewed as a list of URIs or as a virtual sub-tree. The Result node is the virtual sub-tree (beginning at the session base) and the ResultUriList is a LIST of session relative URIs. The actual selection of the nodes must be postponed until either of these nodes (or one of their sub-nodes) is accessed for the first time. Either nodes represent a read-only snapshot that is valid until the end of the session.

It is possible to further refine the selection by specifying the Filter node. The Filter node is an LDAP filter expression or a simple wild card ('*') which selects all the nodes. As the wild card is the default, all nodes selected by the Target are selected by default.

The Filter must be applied to each of the nodes selected by target in the set N. By definition, these nodes are *interior nodes only*. LDAP expressions assert values depending on their *key*. In this case, the child leaf nodes of a node in set N are treated as the property on their parent node.

The attribute name in the LDAP filter can only reference a direct leaf node of the node in the set N or an interior node with the DDF type org.osgi.service.dmt.DmtConstants.DDF_LIST with leaf nodes as children, i.e. a *LIST*. A LIST of primitives must be treated in the filter as a multi valued property, any of its values satisfy an assertion on that attribute.

Attribute names must not contains a slash, that is, it is only possible to assert values directly below the node selected by the target.

Each of these leaf nodes and LISTs can be used in the LDAP Filter as a key/value pair. The comparison must be done with the type used in the Dmt Data object of the compared node. That is, if the

Dmt Admin data is a number, then the comparison rules of the number must be used. The attributes given to the filter must be converted to the Java object that represents their type.

The set N must therefore consists only of nodes where the Filter matches.

It is allowed to change the Target or the Filter node after the results are read. In that case, the Result and ResultUriList must be cleared instantaneously and the search redone once either result node is read.

The initial value of Target is the empty string, which indicates no target.

Table 154.7 Sub-tree Description for Filter

Name	Act	Type	Card.	S	Description
Target	Get Set	string	1	A	An absolute URI always ending in a slash ('/'), with optional wildcards, selecting a set of sub-nodes N. Wildcards can be an asterisk ('*' \u002A) or a minus sign ('-' \u002D). An asterisk can be used in place of a single node name in the URI, a minus sign stands for any number of consecutive node names. The default value of this node is the empty string, which indicates that no nodes must be selected. Changing this value must clear any existing results. If the Result() or ResultUriList is read to get N then a new search must be executed. A URI must always end in '/' to indicate that the target can only select interior nodes.
Filter	Get Set	string	1	A	An optional filter expression that filters nodes in the set N selected by Target. The filter expression is an LDAP filter or an asterisk ('*'). An asterisk is the default value and matches any node in set N. If an LDAP expression is set in the Filter node then the set N must only contain nodes that match the given filter. The values the filter asserts are the immediate leafs and LIST nodes of the nodes in set N. The name of these child nodes is the name of the attribute matched in the filter. The nodes can be removed by the Filter implementation after a timeout defined by the implementation.
Limit	Get Set	integer	1	A	Limits the number of results to the given number. If this node is not set there is no limit. The default value is not set, thus no limit.
Result	Get	NODE	1	A	The Result tree is a virtual read-only tree of all nodes that were selected by the Target and matched the Filter, that is, all nodes in set N. The Result node acts as a parent instead of the session root for each node in N. The Result node is a snapshot taken the first time it is accessed after a change in the Filter and/or the Target nodes.

Name	Act	Type	Card.	S	Description
ResultUriList	Get	LIST	1	A	A list of URIs of nodes in the Device Management Tree from the node selected by the Target that match the Filter node. All URIs are relative to current session. The Result node is a snapshot taken the first time it is accessed after a change in the Filter and/or the Target nodes.
[list]	Get	string	0..*	D	
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

154.8.6 Framework

The Framework node represents the information about the Framework itself. The Framework node allows manipulation of the OSGi framework, start level, framework life cycle, and bundle life cycle.

All modifications to a Framework object must occur in an atomic session. All changes to the framework must occur during the commit.

The Framework node allows the manager to install (create a new child node in Bundle), to uninstall change the state of the bundle (see Bundle.RequestedState()), update the bundle (see URL), start/ stop bundles, and update the framework. The implementation must execute these actions in the following order during the commit of the session:

1. Create a snapshot of the current installed bundles and their state.
2. stop all bundles that will be uninstalled and updated
3. Uninstall all the to be uninstalled bundles (bundles whose RequestedState is Bundle.UNINSTALLED)
4. Update all bundles that have a modified URL with this URL using the Bundle `update(InputStream)` method in the order that the order that the URLs were last set.
5. Install any new bundles from their URL in the order that the order that the URLs were last set.
6. Refresh all bundles that were updated and installed
7. Ensure that all the bundles have their correct start level
8. If the RequestedState was set, follow this state. Otherwise ensure that any Bundles that have the AutoStart flag set to `true` are started persistently. Transiently started bundles that were stopped in this process are not restarted. The bundle id order must be used.
9. Wait until the desired start level has been reached
10. Return from the commit without error.

If any of the above steps runs in an error (except the restart) than the actions should be undone and the system state must be restored to the snapshot.

If the System Bundle was updated (its URL) node was modified, then after the commit has returned successfully, the OSGi Framework must be restarted.

Table 154.8 Sub-tree Description for Framework

Name	Act	Type	Card.	S	Description
StartLevel	Get Set	integer	1	A	The StartLevel manages the Framework's current Start Level. Maps to the Framework Start Level `set/getStartLevel()` methods. This node can set the requested Framework's StartLevel, however it doesn't store the value. This node returns the Framework's StartLevel at the moment of the call.

Name	Act	Type	Card.	S	Description
InitialBundleS-tartLevel	Get Set	integer	1	A	Configures the initial bundle start level, maps to the the FrameworkStartLevel set/getInitialBundleStartLevel() method.
Bundle	Get	MAP	1	A	The MAP of location -> Bundle. Each Bundle is uniquely identified by its location. The location is a string that must be unique for each bundle and can be chosen by the management system.
[String]	Add Get	Bundle	0..*	D	The Bundles node will be automatically filled from the installed bundles, representing the actual state. New bundles can be installed by creating a new node with a given location. At commit, this bundle will be installed from their Bundle.URL node. The location of the System Bundle must be "System Bundle" (see the Core's Constants.SYSTEM_BUNDLE_LOCATION), this node cannot be uninstalled and most operations on this node have special meaning. It is strongly recommended to use a logical name for the location of a bundle, for example reverse domain names or a UUID. To uninstall a bundle, set the Bundle.RequestedState to UNINSTALLED, the nodes in Bundle cannot be deleted.
Property	Get	MAP	1	A	The Framework Properties.
[String]	Get	string	0..*	D	The Framework properties come from the Bundle Context getProperty() method. However, this method does not provide the names of the available properties. If the handler of this node is aware of the framework properties then these should be used to provide the node names. If these properties are now known, the handler must synthesize the names from the following sources • System Properties (as they are backing the Framework properties) • Launching properties as defined in the OSGi Core specification • Properties in the residential specification • Other known properties

154.8.7 Wire

A Wire is a link between two bundles where the semantics of this link is defined by the used name space. This is closely modeled after the Wiring API in the Core Framework.

Table 154.9 Sub-tree Description for Wire

Name	Act	Type	Card.	S	Description
Namespace	Get	string	1	A	The name space of this wire. Can be: • osgi.wiring.bundle - Defined in the OSGi Core • osgi.wiring.package - Defined in the OSGi Core • osgi.wiring.host - Defined in the OSGi Core • osgi.wiring.rmt.service - Defined in this specification • * - Generic name spaces The osgi.wiring.rmt.service name space is not defined by the OSGi Core as it is not part of the module layer. The name space has the following layout: • Requirement - A filter on the service.id service property. • Capability - All service properties as attributes. No defined directives. • Requirer - The bundle that has gotten the service • Provider - The bundle that has registered the service There is a wire for each registration-get pair. That is, if a service is registered by A and gotten by B and C then there are two wires: B->A and C->A.
Requirement	Get	Requirement	1	A	The Requirement that caused this wire.
Capability	Get	Capability	1	A	The Capability that satisfied the requirement of this wire.
Requirer	Get	string	1	A	The location of the Bundle that contains the requirement for this wire.
Provider	Get	string	1	A	The location of the Bundle that provides the capability for this wire.
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

154.8.8 Wire.Capability

Describes a Capability.

Table 154.10 Sub-tree Description for Wire.Capability

Name	Act	Type	Card.	S	Description
Directive	Get	MAP	1	A	The Directives for this capability.
[String]	Get	string	0..*	D	
Attribute	Get	MAP	1	A	The Attributes for this capability.
[String]	Get	string	0..*	D	

154.8.9 Wire.Requirement

Describes a Requirement.

Table 154.11 Sub-tree Description for Wire.Requirement

Name	Act	Type	Card.	S	Description
Filter	Get	string	1	A	The Filter string for this requirement.
Directive	Get	MAP	1	A	The Directives for this requirement. These directives must contain the filter: directive as described by the Core.
[String]	Get	string	0..*	D	
Attribute	Get	MAP	1	A	The Attributes for this requirement.
[String]	Get	string	0..*	D	

154.9 org.osgi.dmt.service.log

154.9.1 Log

Provides access to the Log Entries of the Log Service.

Table 154.12 Sub-tree Description for Log

Name	Act	Type	Card.	S	Description
LogEntries	Get	LIST	1	A	A potentially long list of Log Entries. The length of this list is implementation dependent. The order of the list is most recent event at index 0 and later events with higher consecutive indexes. No new entries must be added to the log when there is an open exclusive or atomic session.
[list]	Get	LogEntry	0..*	D	

154.9.2 LogEntry

A Log Entry node is the representation of a LogEntry from the OSGi Log Service.

Table 154.13 Sub-tree Description for LogEntry

Name	Act	Type	Card.	S	Description
Time	Get	date_time	1	A	Time of the Log Entry.
Level	Get	integer	1	A	The severity level of the log entry. The value is the same as the Log Service level values: • LOG_ERROR 1 • LOG_WARNING 2 • LOG_INFO 3 • LOG_DEBUG 4 Other values are possible because the Log Service allows custom levels.
Message	Get	string	1	A	Textual, human-readable description of the log entry.
Bundle	Get	string	1	A	The location of the bundle that originated this log or an empty string.

Name	Act	Type	Card.	S	Description
Exception	Get	string	0,1	A	Human readable information about an exception. Provides the exception information if any, optionally including the stack trace.

155 TR-157 Amendment 3 Software Module Guidelines

Version 1.0

[1] *Broadband Forum* (BBF) has defined an object model for managing the software modules in a CPE. The BBF Software Modules object defines Execution Environments, Deployment Units, and Execution Units. These concepts are mapped in the following table.

Table 155.1 *Mapping of concepts*

Software Modules Concept	OSGi Concept
Execution Environment	OSGi Framework
Deployment Unit	Bundle
Execution Unit	Bundle

There can be multiple Execution Environments of the same or different types. The parent Execution Environment is either the native environment, for example Linux, or it can be another Framework. A BBF Deployment Unit and Execution Unit both map to a bundle since there is no need to separate those concepts in OSGi. An implementation of this object model should have access to all the Execution Environments as the Deployment Units and Execution Units are represented in a single table.

This section is not a specification in the normal sense. The intention of this chapter is to provide guidelines for implementers of the [4] *TR-157a3 Internet Gateway Device Software Modules* on an OSGi Framework.

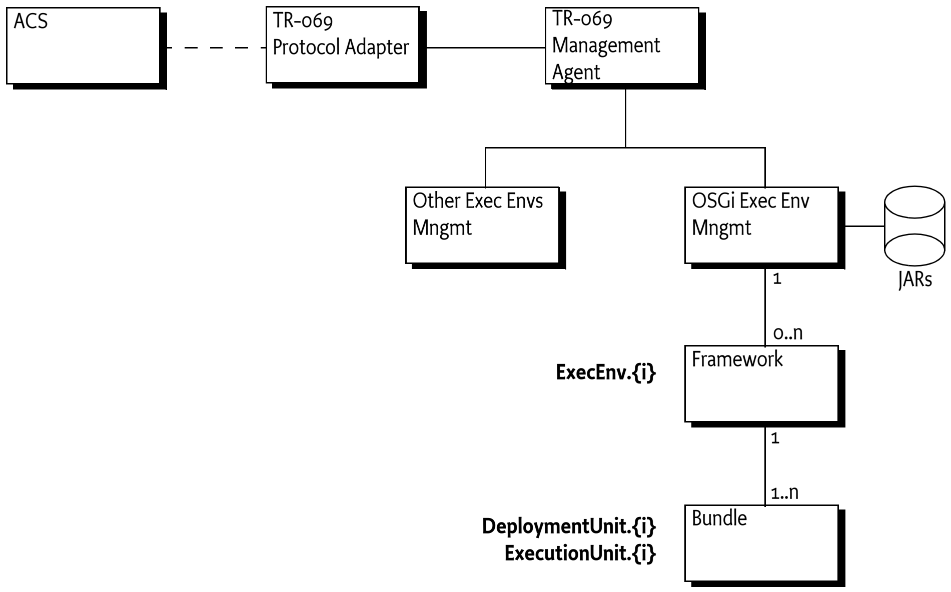
155.1 Management Agent

The Broadband Forum TR-157 Software Modules standard provides a uniform view of the different execution environments that are available in a device. Execution Environments can model the underlying operating system, an OSGi framework, or other environments that support managing the execution of code.

Most parameters in the Software Modules object model map very well to their OSGi counter parts. However, there are a number of issues that require support from a *management agent*. This management agent must maintain state to implement the contract implied by the Software Modules standard. For example, the OSGi Framework does not have an Initial Start Level, an OSGi Framework always starts at an environment property defined start level. However, the standard requires that a Framework must start at a given level after it is launched.

There are many other actions that require a management agent to provide the functionality required by TR-157 that is not build into the OSGi Framework since the standard requires a view that covers the whole device, not just the OSGi environment. The assumed architecture is depicted in Figure 155.1.

Figure 155.1 Management Agent Architecture



155.2 Parameter Mapping

The following table provides OSGi specific information for the different parameters in the Software Modules object model.

Table 155.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	Mapping in case of OSGi
Device.SoftwareModules.	
ExecEnvNumberOfEntries	
DeploymentUnitNumberOfEntries	
ExecutionUnitNumberOfEntries	
Device.SoftwareModules.ExecEnv.{i}.	
Enable	Indicates whether or not this OSGi Framework is enabled. Disabling an enabled OSGi Framework must stop it, while enabling a disabled OSGi Framework must launch it. When an Execution Environment is disabled, Bundles installed in that OSGi Framework will be unaffected, but any Bundles on that OSGi Framework are automatically made inactive. When an OSGi Framework is disabled it is impossible to make changes to the installed bundles, install new bundles, or query any information about the bundles. Disabling the OSGi Framework could place the device in a non-manageable state. For example, if the OSGi Framework runs the Protocol Adapter or has a management agent then it is possible that the device can no longer be restarted.
Status	Indicates the status of the OSGi Framework. Enumeration of: • Up - The OSGi Framework is up and running. • Error - The OSGi Framework could not be launched. • Disabled - The OSGi Framework is not enabled

TR-069 Software Module Object Parameter	Mapping in case of OSGi
`Reset`	Setting this parameter to `true` causes this OSGi Framework to revert back to the state it was in when the device last issued a `0 BOOTSTRAP Inform` event (bootstrap). The following requirements dictate what must happen for the reset to be complete: • The system must restore the set of bundles that were present at the last bootstrap event. That means that installed bundles since that moment must be uninstalled, updated bundles rolled back, and uninstalled bundles re-installed. • The OSGi Framework must roll back to the version it had during the previous rollback. • The OSGi Framework must be restarted after the previous requirements have been met. The value of this parameter is not part of the device configuration and is always `false` when read.
`Alias`	A non-volatile handle used to reference this instance for alias based addressing.
`Name`	A Name that adequately distinguishes this OSGi Framework from all other OSGi Frameworks. This must be the OSGi Framework UUID as stored in the `org.osgi.framework.uuid` property.
`Type`	Indicates the complete type and specification version of this `ExecEnv`. For an OSGi Framework it must be: `OSGi <version>` Where the `<version>` is the value of the framework property `org.osgi.framework.version`
`InitialRunLevel`	The run level that this `ExecEnv` will be in upon startup (whether that is caused by a CPE Boot or the Execution Environment starting). Run levels map to directly OSGi start levels. However, the OSGi Framework has no concept of an initial start level, it can use the `org.osgi.framework.startlevel.beginning` environment property but this requires a management to control it. A management agent must therefore handle this value and instruct the OSGi Framework to move to this start level after a reboot. If the value of `CurrentRunLevel` is set to -1, then the value of this parameter is irrelevant when read. Setting its value to -1 must have no impact on the start level of this OSGi Framework.
`RequestedRunLevel`	Sets the start level of this OSGi Framework, meaning that altering this parameter's value will change the value of the `CurrentRunLevel` asynchronously. Start levels dictate which Bundles will be started. Setting this value when `CurrentRunLevel` is -1 must have no impact on the start Level of this OSGi Framework. The value of this parameter is not part of the device configuration and must always be -1 when read.

TR-069 Software Module Object Parameter	Mapping in case of OSGi
CurrentRunLevel	The start level that this OSGi Framework is currently operating in. This value is altered by changing the RequestedRunLevel parameter. Upon startup (whether that is caused by a CPE Boot or the Execution Environment starting) CurrentRunLevel must be set equal to InitialRunLevel by some management agent. If Run Levels are not supported by this OSGi Framework then CurrentRunLevel must be -1.
Version	The Version of this OSGi Framework as specified by its Vendor. This is not the version of its specification. Must be the value of the System Bundle's getVersion() method.
Vendor	The vendor that produced this OSGi Framework, the value of the org.osgi.framework.vendor Framework property.
ParentExecEnv	The value must be the path name of a row in the ExecEnv table, it can either be the operating system or another OSGi Framework if the framework is nested. If the referenced object is deleted, the parameter value must be set to an empty string. If this value is an empty string then this is the *Primary Execution Environment.*
AllocatedDiskSpace	Implementation specific.
AvailableDiskSpace	Implementation specific.
AllocatedMemory	Implementation specific.
AvailableMemory	Implementation specific.
ProcessorRefList	Comma-separated list of paths into the DeviceInfo.Processor table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the processors that this OSGi Framework has available to it.
ActiveExecutionUnits	Comma-separated list of paths into the ExecutionUnit table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the Bundles currently active on this OSGi Framework.
Device.SoftwareModules. DeploymentUnit.{i}.	This table serves as the Bundles inventory and contains status information about each Bundle. A new instance of this table gets created during the installation of a Bundle.
UUID	A Universally Unique Identifier either provided by the ACS, or generated by the CPE, at the time of Deployment Unit Installation. The format of this value is defined by [2] *RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace* Version 3 (Name-Based) and [5] *TR-069a3 CPE WAN Management Protocol.* This value must not be altered when the Bundle is updated. A management agent should use the UUID as the bundle location since the location plays the same role.
DUID	The Bundle id from the getBundleId() method.
Alias	A non-volatile handle used to reference this instance.
Name	Indicates the Bundle Symbolic Name of this Bundle. The value of this parameter is used in the generation of the UUID based on the rules defined in [5] *TR-069a3 CPE WAN Management Protocol.*

TR-069 Software Module Object Parameter	Mapping in case of OSGi
Status	Indicates the status of this Bundle. Enumeration of: • Installing - This bundle is in the process of being Installed and should transition to the Installed state. This state will never be visible in an OSGi Framework. • Installed - This bundle has been successfully installed.This maps to the Bundle INSTALLED or RESOLVED state. • Updating - This bundle is in the process of being updated and should transition to the Installed state. This state will never be visible in an OSGi Framework. • Uninstalling - This bundle is in the process of being uninstalled and should transition to the uninstalled state.This state will never be visible in an OSGi Framework. • Uninstalled - This bundle has been successfully uninstalled. This state will never be visible in an OSGi Framework.
Resolved	Indicates whether or not this DeploymentUnit has resolved all of its dependencies. Must be true if this Bundle's state is ACTIVE, STARTING, STOPPING, or RESOLVED. Otherwise it must be false.
URL	Contains the URL used by the most recent ChangeDUState RPC to either Install or Update this Bundle. This must be remembered by a management agent since this information is not available in a Bundle.
Description	Textual description of this Bundle, must be the value of the Bundle-Description manifest header or an empty string if not present.
Vendor	The author of this DeploymentUnit formatted as a domain name. The value of this parameter is used in the generation of the UUID based on the rules defined in [5] *TR-069a3 CPE WAN Management Protocol*. The recommended value is the value of the Bundle-Vendor header.
Version	Version of this Bundle, it mist be he value of the geVersion() method.
VendorLogList	Empty String
VendorConfigList	Empty String
ExecutionUnitList	A path into the ExecutionUnit table for the corresponding ExecutionUnit for this Bundle, which is also the bundle since the relation is 1:1.
ExecutionEnvRef	The value must be the path name of a row in the ExecEnv table of the corresponding OSGi Framework.
Device.SoftwareModules. ExecutionUnit.{i}.	This table serves as the Execution Unit inventory and contains both status information about each Execution Unit as well as configurable parameters for each Execution Unit. This list contains all the bundles since in an OSGi Framework Deployment Unit and Execution Unit are mapped to Bundles.
EUID	Table wide identifier for a bundle chosen by the OSGi Framework during installation of the associated DeploymentUnit. The value must be unique across ExecEnv instances. It is recommended that this be a combination of the ExecEnv.{i}.Name and an OSGi Framework local unique value. The unique value for an OSGi framework should be the Bundle Location.
Alias	A non-volatile handle used to reference this instance.
Name	The name should be unique across all Bundles instances contained within its associated DeploymentUnit. As the Deployment Unit and the Execution Unit are the same the value must be the Bundle Symbolic Name.
ExecEnvLabel	The name must be unique across all Bundles contained within a specific OSGi Framework. This must therefore be the Bundle Id.

TR-069 Software Module Object Parameter	Mapping in case of OSGi
AutoStart	If true and the proper start level is met, then this Bundle will be automatically started by the device after its OSGi Framework's start level is met. If false this Bundle must not be started after launch until it is explicitly commanded to do so. An OSGi bundle is persistently started or transiently started. It is not possible to change this state without affecting the active state of the bundle. Therefore, if the AutoStart is set to true, the bundle must be started persistently, even if it is already started. This will record the persistent start state. If the AutoStart is set to false, the bundle must be stopped. Therefore, in an OSGi Framework setting the AutoStart flag to true has the side effect that the bundle is started if it was not active; setting it to false will stop the bundle.
RunLevel	Determines when this Bundle will be started. If AutoStart is true and the CurrentRunLevel is greater than or equal to this RunLevel, then this ExecutionUnit must be started, if run levels are enabled. This maps directly to the Bundles start level.
Status	Indicates the status of this ExecutionUnit. Enumeration of: • Idle - This Bundle is in an Idle state and not running. This maps to the Bundle INSTALLED or Bundle RESOLVED state. • Starting - This Bundle is in the process of starting and should transition to the Active state. This maps to the STARTING state in OSGi. In an OSGi Framework, lazily activated bundles can remain in the STARTING state for a long time. • Active - This instance is currently running. This maps to the Bundle ACTIVE state. • Stopping - This instance is in the process of stopping and should transition to the Idle state.
RequestedState	Indicates the state transition that the ACS is requesting for this Bundle. Enumeration of: • Idle - If this Bundle is currently in STARTING or ACTIVE state then the CPE must attempt to stop the Bundle; otherwise this requested state is ignored. • Active - If this Bundle is currently in the INSTALLED or RESOLVED state the management agent must attempt to start the Bundle. If this ExecutionUnit is in the STOPPING state the request is rejected and a fault raised. Otherwise this requested state is ignored. If this Bundle is disabled and an attempt is made to alter this value, then a CWMP Fault must be generated. The value of this parameter is not part of the device configuration and is always an empty string when read. Bundles must be started transiently when the AutoStart is false, otherwise persistently.

TR-069 Software Module Object Parameter	Mapping in case of OSGi
ExecutionFaultCode	If while running or transitioning between states this Bundle raises an Exception then this parameter embodies the problem. Enumeration of: • NoFault - No fault, default value. • FailureOnStart - Threw an exception when started. • FailureOnAutoStart - Failed to be started by the framework, this must be intercepted by the management agent because this is a Framework Error event. • FailureOnStop - Raised an exception while stopping • FailureWhileActive - Raised when a bundle cannot be restarted after a background operation of the Framework, for example refreshing. • DependencyFailure - Failed to resolve • UnStartable - Cannot be raised in OSGi since this is the same error as FailureOnStart. For fault codes not included in this list, the vendor can include vendor-specific values, which must use the format defined in Section 3.3 of [6] *TR-106a4 Data Model Template for TR-069-Enabled Devices*.
ExecutionFaultMessage	If while running or transitioning between states this Bundle identifies a fault this parameter provides a more detailed explanation of the problem enumerated in the ExecutionFaultCode. If ExecutionFaultCode has the value of NoFault then the value of this parameter must be an empty string and ignored. This message must be the message value of the exception thrown by the Bundle.
Vendor	Vendor of this Bundle. The value of the Bundle-Vendor manifest header
Description	Textual description of this Bundle. The value of the Bundle-Description manifest header
Version	Version of the Bundle. The value of the getVersion() method.
VendorLogList	Empty string.
VendorConfigList	Empty string.
DiskSpaceInUse	Implementation defined
MemoryInUse	Implementation defined
References	Empty String
AssociatedProcessList	Empty String as an OSGi bundle reuses the process of the VM.
SupportedDataModelList	Comma-separated list of strings. Each list item must be the path name of a row in the DeviceInfo.SupportedDataModel table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the CWMP-DT schema instances that have been introduced to this device because of the existence of this ExecutionUnit. In OSGi this is implementation defined.
ExecutionEnvRef	The path to the OSGi Framework that hosts this bundle in the ExecEnv table.
Device.SoftwareModules. ExecutionUnit.{i}.Extensions.	This object proposes a general location for vendor extensions specific to this Execution Unit, which allows multiple Execution Units to expose parameters without the concern of conflicting parameter names. This part is not used in OSGi.

155.3 References

[1] *Broadband Forum*
https://www.broadband-forum.org

[2] *RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace*
https://www.ietf.org/rfc/rfc4122.txt

[3] *TR-157a3 Component Objects for CWMP*
https://www.broadband-forum.org/technical/download/TR-157_Amendment-3.pdf

[4] *TR-157a3 Internet Gateway Device Software Modules*
https://www.broadband-forum.org/cwmp/tr-157-1-3-0-igd.html#D.InternetGatewayDevice.SoftwareModules

[5] *TR-069a3 CPE WAN Management Protocol*
https://www.broadband-forum.org/technical/download/TR-069_Amendment-3.pdf

[6] *TR-106a4 Data Model Template for TR-069-Enabled Devices*
https://www.broadband-forum.org/technical/download/TR-106_Amendment-4.pdf

157 Typed Event Service Specification

Version 1.0

157.1 Introduction

Eventing systems are a common part of software programs, used to distribute information between parts of an application. To address this, the *Event Admin Service Specification* on page 341 was created as one of the earliest specifications defined by the OSGi Compendium. The design and usage of the Event Admin specification, however, makes certain trade-offs that do not fit well with modern application design:

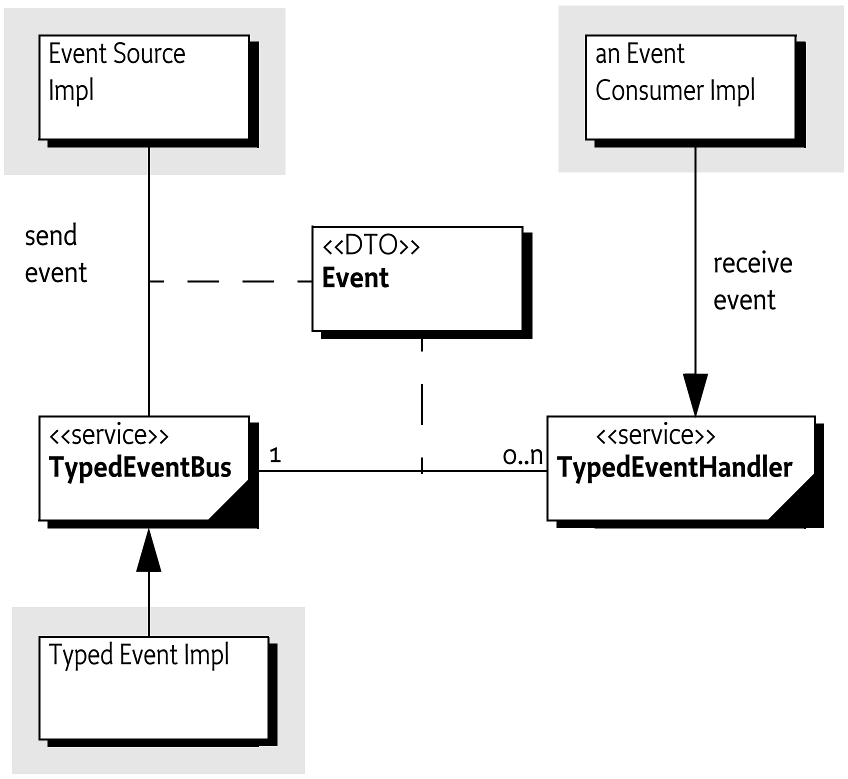
- *Type Safety* - Events are sent and received as opaque maps of key-value pairs. The “schema” of an event is therefore ill-defined and relies on “magic strings” being used correctly to locate data, and on careful handling of data values with unknown types.
- *Unhandled Events* - Events that are sent but have no interested Event Consumers are silently discarded. There is no way to know that an event was not handled, short of disrupting the system by registering a handler for *all* events.
- *Observability* - There is no simple, non-invasive way to monitor the flow of events through the system. The ability to monitor and profile applications using Event Admin is therefore relatively limited.

Adding these features to the original *Event Admin Service Specification* on page 341 specification is not feasible without breaking backward compatibility for clients. Therefore this specification exists to provide an alternative eventing model which supports these different requirements by making different design trade-offs.

157.1.1 Essentials

- *Event* - A set of data created by an Event Source, encapsulated as an object and delivered to one or more Event Consumers.
- *Event Schema* - A definition of the expected data layout within an event, including the names of data fields and the types of data that they contain.
- *Event Topic* - A String identifying the *topic* of an Event, effectively defining the Event Schema and the purpose of the event.
- *Event Source* - A software component which creates and sends events.
- *Event Consumer* - A software component which receives events.
- *DTO* - A Data Transfer Object as per the OSGi DTO Specification.
- *Event Bus* - A software component used by an Event Source and responsible for delivering Events to Event Consumers.

Figure 157.1 *Class and Service overview*



157.1.2 Entities

- *Typed Event Bus* - A service registered by the Typed Event implementation that can be passed an Event object and that will distribute that event to any suitable Event Handler Services.
- *Event Handler* - A service registered by an Event Consumer suitable for receiving Event data from the Typed Event Bus.

157.2 Events

In this specification an Event is a set of string keys associated with data values. The defined set of allowable keys and permitted value types for the keys in an Event is known as the Event Schema. Both the Event Source and Event Consumers must agree on a schema, or set of compatible schemas, in order for events to be consumed correctly.

157.2.1 Type Safe Events

A Type Safe Event is one in which the Event Schema is defined as a Java class. Using a Java class provides a formal definition of the schema - event data uses field names in the class as the keys, and each field definition defines the permitted type of the value.

Type Safe Event classes are expected to conform to OSGi DTO rules. The architecture of OSGi DTOs is described in *OSGi Core Release 8*. All methods, all static fields, and any non public instance fields of an event object must be ignored by the Typed Event Service when processing the Event data.

Some implementations of the Typed Event Service may support Type Safe Event classes that do not conform to the DTO rules, transforming them as needed in an implementation specific way. This is permitted by this specification, however consumers which rely on this behaviour may not be portable between different implementations of this specification.

157.2.1.1 Nested Data Structures

OSGi DTOs are permitted to have data values which are also DTOs, allowing nested data structures to be created. This is also allowed for Type Safe Events, but with the same restriction that the event data must be a tree. There is no restriction on the depth of nesting permitted.

157.2.2 Untyped Events

An Untyped Event is one in which there is no Java class defining the Event Schema. In this case the event data is defined using a `Map` type with `String` keys and values limited to types acceptable as fields in a DTO, excepting:

- DTO types - an untyped event may not have DTOs inside it as these form part of a typed schema.
- Maps are only permitted if they follow the rules for Untyped events, that is having `String` keys and DTO restricted value types excluding DTOs.

Untyped Event instances are capable of representing exactly the same data as present in a Type Safe Event instance, and are also subject to the same restrictions, that is the data must be a tree. Nested data should be included as sub-maps within the event map, and these sub-maps may in turn contain nested data.

157.2.3 Non Standard Type Safe Events

Some Event schemas may be represented by an existing type which does not match the OSGi DTO rules. In this case there are two main options:

- Create a DTO representation of the event schema, and convert from the existing type into the DTO representation in code.
- Convert the event data into an Untyped Event representation using nested Maps.

For example, the following code demonstrates how an object following the JavaBeans pattern can be converted into a DTO type or an untyped map:

```
public class ExampleJavaBean {
    private String message;

    public String getMessage() { return message; }

    public void setMessage(String message) { this.message = message; }
}

public class ExampleEvent {
    public String message;
}

@Component
public class ExampleEventSource {
    private ExampleEvent createEventFromJavaBean(ExampleJavaBean bean) {
        return Converters.standardConverter().convert(bean)
                .to(ExampleEvent.class);
    }

    private Map<String, Object> createMapFromJavaBean(ExampleJavaBean bean) {
        return Converters.standardConverter().convert(bean)
                .to(new TypeReference<Map<String, Object>>(){});
    }
}
```

157.2.4 Event Mutability and Thread Safety

The Typed Event Service is inherently multi-threaded. Events may be published from multiple threads, and event data may be delivered to consumers on multiple threads. Event Sources and Event Consumers must therefore assume that event data is shared between threads from the moment that it is first passed to the `TypedEventBus`.

157.2.4.1 Typed Event Mutability

Typed Events, and in particular DTO types, provide a simple yet powerful mechanism for defining an Event Schema in a type-safe way. However their use of mutable public fields means that they are potentially dangerous when shared between threads. Event Sources and Event Consumers should assume that their event instances are shared between threads and therefore not mutate the event data after publication or receipt.

If an Event Handler does need to make changes to an incoming event then it must copy the event data into a new DTO instance. Note that any nested DTO values in the event data must also be copied if they are to be mutated.

157.2.4.2 Untyped Event Mutability

When an event source publishes untyped event data, it passes a Map instance to the Typed Event Bus. The Typed Event Bus is not required to take a copy of this Map, and therefore the event source must not change the Map, or any data structures within the Map, after the call to `deliverUntyped(String,Map)`.

Untyped Events are delivered as implementations of the Map interface. Bundles consuming untyped events should not rely on the event object being any particular implementation of Map, and should treat the event object as immutable. The Typed Event Bus implementation may make copies of the event data, or enforce the immutability of the map, before passing the event data to an Event Handler.

157.3 Publishing Events

To publish an event, the Event Source must retrieve the Typed Event Bus service from the OSGi service registry. The Event Source then creates an event object and calls one of the Typed Event Bus service's methods to publish the event. Event publication is asynchronous, meaning that when a call to the Typed Event Bus returns there is no guarantee that all, or even any, listeners have been notified.

157.3.1 Event Topics

Events are always published to a topic. The topic of an event defines the *schema* of the event. Topics exist in order to give Event Consumers information about the schema of the event, and the opportunity to register for just the events they are interested in. When a topic is designed, its name should not include any other information, such as the publisher of the event or the data associated with the event, those parts are intended to be stored in the event properties.

The topic therefore serves as a first-level filter for determining which handlers should receive the event. Typed Event service implementations use the structure of the topic to optimize the dispatching of the events to the handlers. The following example code demonstrates how to send an event to a topic.

```
public class ExampleEvent {
    public String message;
}

@Component
public class ExampleEventSource {
    @Reference
    TypedEventBus bus;

    public void sendEvent() {
        ExampleEvent event = new ExampleEvent();
        event.message = "The time is " + LocalDateTime.now();

        bus.deliver("org/osgi/example/ExampleEvent", event);
    }
}
```

Topics are arranged in a hierarchical namespace. Each level is defined by a token and levels are separated by solidi ('/' \u002F). More precisely, the topic must conform to the following grammar:

```
// For further information see General Syntax Definitions in Core

topictoken  :: ( jletterordigit | '-' ) +

topic       ::= topictoken ( '/' topictoken ) *
```

Topics should be designed to become more specific when going from left to right. Consumers can provide a prefix that matches a topic, using the preferred order allows a handler to minimize the number of prefixes it needs to register.

Topics are case-sensitive. As a convention, topics should follow the reverse domain name scheme used by Java packages to guarantee uniqueness. The separator must be a solidus ('/' \u002F) instead of the full stop ('.' \u002E).

This specification uses the convention fully/qualified/package/ClassName/ACTION. If necessary, a pseudo-class-name is used.

157.3.2 Automatically Generated Topics

In many cases the name of a topic contains no information other than defining the schema of the events sent on that topic. Therefore, when publishing a Typed Event to the Typed Event Bus, the Typed Event implementation is able to automatically generate a topic name based on the the type of the event object being published.

For the deliver(Object) method on the Typed Event Bus where no topic string is provided, the implementation must create a topic string using the fully qualified class name of the event object. To convert the class name into a valid topic the full stop . separators must be converted into solidus / separators. A non-normative example implementation follows:

```
public void deliver(Object event) {
    String topicName = event.getClass().getName().replace('.', '/');

    this.deliver(topicName, event);
}
```

The following example demonstrates how an Event Source can make use of an automatically generated topic name.

```
package org.osgi.example;
```

```
public class ExampleEvent {
    public String message;
}

@Component
public class ExampleEventSource {
    @Reference
    TypedEventBus bus;

    public void sendEvent() {
        ExampleEvent event = new ExampleEvent();
        event.message = "The time is " + LocalDateTime.now();

        // This event will be delivered to the
        // topic "org/osgi/example/ExampleEvent"
        bus.deliver(event);
    }
}
```

157.3.3 Thread Safety

The TypedEventBus implementation must be thread safe and allow for simultaneous event publication from multiple threads. For any given source thread, events must be delivered in the same order as they were published by that thread. Events published by different threads, however, may be delivered in a different order from the one in which they were published.

For example, if thread *A* publishes events *1*, *2* and *3*, while thread *B* publishes events *4*, *5* and *6*, then the events may be delivered:

- *1*, *2*, *3*, *4*, *5*, *6*
- *4*, *1*, *2*, *5*, *6*, *3*
- and so on

but events will never be delivered *1*, *2*, *6*, *4*, *5*, *3*

157.4 Receiving Events

Event Consumers can receive events by registering an appropriate Event Handler service in the Service Registry. This is a TypedEventHandler to receive events as type-safe objects, or an UntypedEventHandler to receive events as untyped Map structures.

Published events are then delivered, using the whiteboard pattern, to any Event Handler service which has registered interest in the topic to which the event was published.

157.4.1 Receiving Typed Events

Typed Events are received by registering a TypedEventHandler implementation. This service has a single method notify which receives the String topic name and Object event data. The TypedEventHandler implementation must be registered as a service in the service registry using the TypedEventHandler interface.

The TypedEventHandler interface is parameterized, and so it is expected that the implementation reifies the type parameter into a specific type. In this case the Typed Event implementation must adapt the Event object into the type defined by the TypedEventHandler implementation. Implementations of this specification are free to choose their own adaptation mechanism, however it must guarantee at least the same functionality as *Converter Specification* on page 1457.

A simple example of receiving a typed event follows:

```
public class ExampleEvent {
    public String message;
}

@Component
public class ExampleTypedConsumer implements TypedEventHandler<ExampleEvent> {
    @Override
    public void notify(String topic, ExampleEvent event) {
        System.out.println("Received event: " + event.message);
    }
}
```

If the TypedEventHandler implementation is unable to reify the type, or the required type is more specific than the reified type, then the Typed Event Handler must be registered with the event.type service property. This property has a string value containing the fully-qualified type name of the type that the Typed Event Handler expects to receive. This type must be loaded by the Typed Event implementation using the classloader of the bundle which registered the Typed Event Handler service. The loaded type must then be used as the target type when converting events. For example:

```
public class ExampleEvent {
    public String message;
}

public class SpecialisedExampleEvent extends ExampleEvent {
    public int sequenceId = Integer.MIN_VALUE;
}

@Component
@EventType(SpecialisedExampleEvent.class)
public class ExampleTypedConsumer implements TypedEventHandler<ExampleEvent> {
    @Override
    public void notify(String topic, ExampleEvent event) {
        System.out.println("Received event: " + event.message);

        // The event will always be of type SpecialisedExampleEvent
        System.out.println("Event sequence id was " +
            ((SpecialisedExampleEvent) event).sequenceId);
    }
}
```

By default the reified type of the TypedEventHandler will be used as the target topic for the Event Handler. If the event.type property is set then this is used as the default topic instead of the reified type. To use a specific named topic the Typed Event Handler service may be registered with an event.topics service property specifying the topic(s) as a String+ value.

```
public class ExampleEvent {
    public String message;
}

@Component
@EventTopics({"foo", "foo/bar"})
public class ExampleTypedConsumer implements TypedEventHandler<ExampleEvent> {
    @Override
    public void notify(String topic, ExampleEvent event) {
        System.out.println("Event received on topic: " + topic +
                " with message: " + event.message);
    }
}
```

157.4.2 Receiving Untyped Events

Untyped Events are received by registering an UntypedEventHandler implementation. This service has a single method notifyUntyped which receives the String topic name and Map event data. The Untyped Event Handler implementation must be registered as a service in the service registry using the UntypedEventHandler interface.

When delivering an event to an Untyped Event Handler the Typed Event Service must, if necessary, convert the event data to a nested map structure.

The event.topics service property must be used when registering an Untyped Event Hander service. If it is not, then no events will be delivered to that Untyped Event Handler service.

```
public class ExampleEvent {
    public String message;
}

@Component
@EventTopics({"foo", "foo/bar"})
public class ExampleUntypedConsumer implements UntypedEventHandler {
    @Override
    public void notifyUntyped(String topic, Map<String,Object> event) {
        System.out.println("Event received on topic: " + topic
                + " with message: " + event.get("message"));
    }
}
```

157.4.3 Wildcard Topics

The event.topics property may contain one or more wildcard topics. These are Strings which contain a topic name and append "/*". This value means that the Event Handler must be called for Events sent to sub-topics of the named topic. For example the component:

```
@Component
@EventTopics("foo/*")
public class ExampleUntypedConsumer implements UntypedEventHandler {
    @Override
    public void notifyUntyped(String topic, Map<String,Object> event) {
        System.out.println("Event received on topic: " + topic
                + " with message: " + event.get("message"));
    }
}
```

would receive events sent to the topics foo/bar and foo/baz, but not the topics foo or foobar/fizzbuzz.

The * character in a wildcard topic must always follow a solidus / character, and must be the final character in the topic string, meaning that topic names such as foo* and foo/*/bar are not valid. The only exception to this rule is that it is valid to use the topic name * to receive events on *all* topics. While it is valid to do so, using the topic * is not typically recommended. For a mechanism to monitor the events flowing through the system see *Monitoring Events* on page 1343.

157.4.4 Unhandled Events

Unhandled Events are events sent by an Event Source but which have no Event Handler service listening to their topic. Rather than these events being discarded, the Typed Event implementation will search the service registry for services implementing UnhandledEventHandler.

If any services are found then the Typed Event implementation will call the notifyUnhandled method passing the topic name and event data to all of the registered Unhandled Event Handler services.

```
public class ExampleEvent {
    public String message;
}

@Component
public class ExampleUnhandledConsumer implements UnhandledEventHandler {
    @Override
    public void notifyUnhandled(String topic, Map<String,Object> event) {
        System.out.println("Unhandled Event received on topic: " + topic);
    }
}
```

157.4.5 Filtering Events

Sometimes the use of a topic is insufficient to restrict the events received by an event consumer. In these cases the consumer can further restrict the events that they receive by using a filter. The filter is supplied using the event.filter service property, the value of which is an LDAP filter string. This filter is applied to the event data, and only events which match the filter are delivered to the event handler service.

157.4.5.1 Nested Event Data

Complex events may contain nested data structures, such as DTOs, as values in the event data. As LDAP filtering is only designed to match against simple data this means that some event properties cannot be filtered using the event.filter property. The event filter is therefore only suitable for use in matching top-level event properties.

157.4.5.2 Ignored Events

Note that the use of a filter is different from receiving an event and choosing to ignore it based on its data. If an event fails to match the filter supplied by an event handler service then it is *not delivered* to that event handler. This means that the event data remains eligible to be sent to an UnhandledEventHandler unless another event handler does receive it. An event that is received, but ignored, by an event handler service *does* count as having been delivered, and so will never be sent to an UnhandledEventHandler.

157.4.6 Failing Event Handlers

Event Handler implementations are called by the Typed Event Bus implementation, and are expected:

- Not to throw exceptions from their callback method
- To return quickly - any long running tasks should be moved to another thread

If a Typed Event Bus implementation detects an Event Handler that is behaving incorrectly, either by throwing exceptions, or by taking a long time to process the event, or some other problem, then the implementation may block further event delivery to that Event Handler.

If an Event Handler is blocked by the event implementation then this situation must be logged. Also, if a blocked Event Handler service is updated then the block must be removed by the implementation. If the updated service continues to behave incorrectly then the block may be reinstated.

157.4.7 Event Handler Service Properties

The service properties that can be used to configure an Event Handler service are outlined in the following table.

Table 157.1 Service properties applicable to Event Handler services

Service Property Name	Type	Description
event.topics	String+	Declares the topic pattern(s) for which the service should be called. This service property is *required* for UntypedEventHandler services, but TypedEventHandler services may omit it if they are only interested in the default topic name for their reified type. See TYPED_EVENT_TOPICS.
event.type	String	Defines the target type into which events should be converted before being passed to the Event Handler service. This service property is *forbidden* for UntypedEventHandler services, but TypedEventHandler services may use it if they wish to further refine the type of data they wish to receive. See TYPED_EVENT_TYPE.
event.filter	String	Defines an LDAP filter which should be tested against the properties in the event data. Only events which pass the filter will be passed to the the Event Handler service. Ths service property is permitted for both TypedEventHandler and UntypedEventHandler services. See TYPED_EVENT_FILTER.

157.4.8 Error Handling

There are several possible error scenarios for Event Handlers:

- TypedEventHandler - If the target event type is not discoverable, that is there is no reified type information, nor is there an event.type property, then the target type for the event is not known. In this situation there is no way for the Typed Event implementation to correctly target an event schema, and the TypedEventHandler must be ignored. The implementation must write a message to the log indicating which service is being ignored.
- TypedEventHandler - If the target event type is discoverable but cannot be loaded using the class loader of the bundle which registered the Typed Event Handler service then there is no way for the Typed Event implementation to correctly target an event schema, and the Event Handler must be ignored. The implementation must write a message to the log indicating which service is being ignored.
- All Handler Types - If the event data cannot be adapted to the target type, that is the incoming data cannot be transformed due to badly mismatched property names or values, then that specific Event cannot be submitted to the Handler. The Typed Event implementation must write a message to the log indicating which service failed to receive the event. If this error occurs repeatedly then the Typed Event implementation may choose to deny list and ignore the Event Handler service. Deny listing decisions must be written to the log.
- All Handler Types - If the event.topics property contains one or more invalid values then the Event Handler service must be ignored. The implementation must write a message to the log indicating which service is being ignored.

157.5 The Typed Event Bus Service

The Typed Event implementation must register a Typed Event Bus service in the service registry. This service must implement and advertise the TypedEventBus interface.

157.5.1 Error Handling

It is not possible to know that an Event cannot be delivered until delivery is attempted. It is therefore not possible (or acceptable, given the asynchronous nature of delivery) to throw an exception to the sender of an event if there are problems delivering the event. The Event Bus service should not throw exceptions from any publication methods except:

- NullPointerException if the event data is null.
- IllegalArgumentException if a topic name is supplied and it violates the topic name syntax.

157.6 Monitoring Events

An important part of a software system is the ability to monitor it appropriately to determine whether it is functioning correctly, without having the measurements disrupt the system. To this end the Typed Event implementation must register a TypedEventMonitor service which can be used to monitor the flow of events through the Event Bus.

Events flowing through the Typed Event Bus can be monitored using one of the monitorEvents methods from the TypedEventMonitor service. These methods return a PushStream which delivers MonitorEvent instances each time an event is sent via the TypedEventBus. The monitor events contain the event topic, the event data, and a timestamp indicating when the event was sent.

157.6.1 Event History

In a running system it is often useful for monitoring tools to replay recent data immediately after a problem has occurred. For that reason Typed Event Monitor instances may store past events so that they can be replayed if requested. There are two monitorEvents methods capable of replaying history:

- monitorEvents(int) takes an int representing the number of past events that should be replayed from the cached history
- monitorEvents(Instant) takes an Instant, representing the time in the past from which the stream of monitoring events should start.

Note that storing Event History is considered a best-effort option and it is not required that the implementation supply the full set of requested events. If insufficient past events are available then the implementation must provide the maximum amount of history available.

157.7 Capabilities

157.7.1 osgi.implementation Capability

The Typed Event implementation bundle must provide the osgi.implementation capability with the name TYPED_EVENT_IMPLEMENTATION. This capability can be used by provisioning tools and during resolution to ensure that a Typed Event implementation is present. The capability must also declare a uses constraint for the org.osgi.service.typedevent package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
 osgi.implementation="osgi.typedevent";
 uses:="org.osgi.service.typedevent";
 version:Version="1.0"
```

The RequireTypedEvent annotation can be used to require this capability.

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 711.

157.7.2 osgi.service Capability

The bundle providing the Typed Event Bus service must provide capabilities in the osgi.service namespace representing the services it is required to register. This capability must also declare uses constraints for the relevant service packages:

```
Provide-Capability: osgi.service;
 objectClass:List<String>="org.osgi.service.typedevent.TypedEventBus";
 uses:="org.osgi.service.typedevent",
 osgi.service;
 objectClass:List<String>="org.osgi.service.typedevent.monitor.TypedEventMonitor";
 uses:="org.osgi.service.typedevent.monitor"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

157.8 Security

157.8.1 Topic Permission

The TopicPermission class allows fine-grained control over which bundles may post events to a given topic and which bundles may receive those events.

The target parameter for the permission is the topic name. TopicPermission classes uses a wildcard matching algorithm similar to the BasicPermission class, except that solidi ('/' \u002F) are used as separators instead of full stop characters. For example, a name of a/b/* implies a/b/c but not x/y/z or a/b.

There are two available actions: PUBLISH and SUBSCRIBE. These control a bundle's ability to either publish or receive events, respectively. Neither one implies the other.

157.8.2 Required Permissions

Bundles that need to consume events must be granted permission to register the appropriate handler service. For Example: ServicePermission[org.osgi.service.typedevent.TypedEventHandler, REGISTER] or ServicePermission[org.osgi.service.typedevent.UntypedEventHandler, REGISTER] or ServicePermission[org.osgi.service.typedevent.UnhandledEventHandler, REGISTER]. In addition, bundles that consume events require TopicPermission[<topic>, SUBSCRIBE] for each topic they want to be notified about.

Bundles that need to publish events must be granted permission to get the TypedEventBus service, that is ServicePermission[org.osgi.service.typedevent.TypedEventBus, GET] so that they may retrieve the Typed Event Bus and use it. In addition, event sources require TopicPermission[<topic>, PUBLISH] for each topic they want to send events to. This includes any default topic names that are used when publishing

Bundles that need to monitor events flowing through the bus must be granted permission to get the TypedEventMonitor service, that is ServicePermission[org.osgi.service.typedevent.monitor.TypedEventMonitor, GET] so that they may retrieve the Typed Event Monitor and use it.

Only a bundle that provides a Typed Event implementation should be granted ServicePermission[org.osgi.service.typedevent.TypedEventBus, REGISTER] and ServicePermission[org.osgi.service.typedevent.monitor.TypedEventMonitor, REGISTER] to register the services defined by this specification.

The Typed Event implementation must be granted ServicePermission[org.osgi.service.typedevent.TypedEventHandler, GET], ServicePermission[org.osgi.service.typedevent.UntypedEventHandler, GET], ServicePermission[org.osgi.service.typedevent.UnhandledEventHandler, GET], ServicePermission[org.osgi.service.typedevent.TypedEventBus, REGISTER] and ServicePermission[org.osgi.service.typedevent.monitor.TypedEventMonitor, REGISTER] as these actions are all required to implement the specification.

157.8.3 Security Context During Event Callbacks

During an event notification, the Typed Event implementation's Protection Domain will be on the stack above the handler's Protection Domain. Therefore, if a handler needs to perform a secure operation using its own privileges, it must invoke the doPrivileged method to isolate its security context from that of its caller.

The event delivery mechanism must not wrap event notifications in a doPrivileged call.

157.9 org.osgi.service.typedevent

Typed Event Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.typedevent; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.typedevent; version="[1.0,1.1)"

157.9.1 Summary

- TopicPermission - A bundle's authority to publish or subscribe to typed events on a topic.
- TypedEventBus - The Typed Event service.
- TypedEventConstants - Defines standard names for Typed Event properties.
- TypedEventHandler - Listener for Typed Events.
- UnhandledEventHandler - Listener for Unhandled Events.
- UntypedEventHandler - Listener for Untyped Events.

157.9.2 public final class TopicPermission extends Permission

A bundle's authority to publish or subscribe to typed events on a topic.

A topic is a slash-separated string that defines a topic.

For example:

```
 org / osgi / service / foo / FooEvent / ACTION
```

Topics may also be given a default name based on the event type that is published to the topic. These use the fully qualified class name of the event object as the name of the topic.

For example:

```
com.acme.foo.event.EventData
```

TopicPermission has two actions: publish and subscribe.

Concurrency Thread-safe

157.9.2.1 public static final String PUBLISH = "publish"

The action string publish.

157.9.2.2 public static final String SUBSCRIBE = "subscribe"

The action string subscribe.

157.9.2.3 public TopicPermission(String name, String actions)

name Topic name.

actions publish,subscribe (canonical order).

□ Defines the authority to publish and/or subscribe to a topic within the Typed Event service specification.

The name is specified as a slash-separated string. Wildcards may be used. For example:

```
    org/osgi/service/fooFooEvent/ACTION
    com/isv/*
    *
```

A bundle that needs to publish events on a topic must have the appropriate TopicPermission for that topic; similarly, a bundle that needs to subscribe to events on a topic must have the appropriate TopicPermssion for that topic.

157.9.2.4 public boolean equals(Object obj)

obj The object to test for equality with this TopicPermission object.

□ Determines the equality of two TopicPermission objects. This method checks that specified TopicPermission has the same topic name and actions as this TopicPermission object.

Returns true if obj is a TopicPermission, and has the same topic name and actions as this TopicPermission object; false otherwise.

157.9.2.5 public String getActions()

□ Returns the canonical string representation of the TopicPermission actions.

Always returns present TopicPermission actions in the following order: publish,subscribe.

Returns Canonical string representation of the TopicPermission actions.

157.9.2.6 public int hashCode()

□ Returns the hash code value for this object.

Returns A hash code value for this object.

157.9.2.7 public boolean implies(Permission p)

p The target permission to interrogate.

□ Determines if the specified permission is implied by this object.

This method checks that the topic name of the target is implied by the topic name of this object. The list of TopicPermission actions must either match or allow for the list of the target object to imply the target TopicPermission action.

```
x/y/*,"publish" -> x/y/z,"publish" is true
*,"subscribe" -> x/y,"subscribe"   is true
*,"publish" -> x/y,"subscribe"     is false
x/y,"publish" -> x/y/z,"publish"   is false
```

Returns true if the specified TopicPermission action is implied by this object; false otherwise.

157.9.2.8 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing TopicPermission objects.

Returns A new PermissionCollection object.

157.9.3 public interface TypedEventBus

The Typed Event service. Bundles wishing to publish events must obtain this service and call one of the event delivery methods.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

157.9.3.1 public void deliver(Object event)

event The event to send to all listeners which subscribe to the topic of the event.

□ Initiate asynchronous, ordered delivery of an event. This method returns to the caller before delivery of the event is completed. Events are delivered in the order that they are received by this method.

The topic for this event will be automatically set to the fully qualified type name for the supplied event object.

Logically equivalent to calling deliver(event.getClass().getName().replace('.', '/'), event)

Throws NullPointerException– if the event object is null

157.9.3.2 public void deliver(String topic, Object event)

topic The topic to which this event should be sent.

event The event to send to all listeners which subscribe to the topic.

□ Initiate asynchronous, ordered delivery of an event. This method returns to the caller before delivery of the event is completed. Events are delivered in the order that they are received by this method.

Throws NullPointerException– if the event object is null

IllegalArgumentException– if the topic name is not valid

157.9.3.3 public void deliverUntyped(String topic, Map<String, ?> event)

topic The topic to which this event should be sent.

event A Map representation of the event data to send to all listeners which subscribe to the topic.

□ Initiate asynchronous, ordered delivery of event data. This method returns to the caller before delivery of the event is completed. Events are delivered in the order that they are received by this method.

Throws NullPointerException– if the event map is null

IllegalArgumentException– if the topic name is not valid

157.9.4 public final class TypedEventConstants

Defines standard names for Typed Event properties.

Provider Type Consumers of this API must not implement this type

157.9.4.1 public static final String TYPED_EVENT_FILTER = "event.filter"

The name of the service property used to indicate a filter that should be applied to events from the TYPED_EVENT_TOPICS. Only events which match the filter will be delivered to the Event Handler service.

If this service property is not present then all events from the topic(s) will be delivered to the Event Handler service.

157.9.4.2 public static final String TYPED_EVENT_IMPLEMENTATION = "osgi.typedevent"

The name of the implementation capability for the Typed Event specification

157.9.4.3 public static final String TYPED_EVENT_SPECIFICATION_VERSION = "1.0"

The version of the implementation capability for the Typed Event specification

157.9.4.4 public static final String TYPED_EVENT_TOPICS = "event.topics"

The name of the service property used to indicate the topic(s) to which an a TypedEventHandler or UntypedEventHandler service is listening.

If this service property is not present then the reified type parameter from the TypedEventHandler implementation class will be used to determine the topic.

157.9.4.5 public static final String TYPED_EVENT_TYPE = "event.type"

The name of the service property used to indicate the type of the event objects received by a TypedEventHandler service.

If this service property is not present then the reified type parameter from the TypedEventHandler implementation class will be used.

157.9.5 public interface TypedEventHandler<T>

<T> The type of the event to be received

Listener for Typed Events.

TypedEventHandler objects are registered with the Framework service registry and are notified with an event object when an event is sent.

TypedEventHandler objects are expected to reify the type parameter T with the type of object they wish to receive when implementing this interface. This type can be overridden using the TypedEventConstants.TYPED_EVENT_TOPICS service property.

TypedEventHandler objects may be registered with a service property TypedEventConstants.TYPED_EVENT_TOPICS whose value is the list of topics in which the event handler is interested.

For example:

```
String[] topics = new String[] {
  "com/isv/*"
};
Hashtable ht = new Hashtable();
```

```
ht.put(EventConstants.TYPE_SAFE_EVENT_TOPICS, topics);
context.registerService(TypedEventHandler.class, this, ht);
```

Concurrency Thread-safe

157.9.5.1 public void notify(String topic, T event)

topic The topic to which the event was sent

event The event that occurred.

□ Called by the TypedEventBus service to notify the listener of an event.

157.9.6 public interface UnhandledEventHandler

Listener for Unhandled Events.

UnhandledEventHandler objects are registered with the Framework service registry and are notified with an event object when an event is sent, but no other handler is found to receive the event

Concurrency Thread-safe

157.9.6.1 public void notifyUnhandled(String topic, Map<String, Object> event)

topic The topic to which the event was sent

event The event that occurred.

□ Called by the TypedEventBus service to notify the listener of an unhandled event.

157.9.7 public interface UntypedEventHandler

Listener for Untyped Events.

UntypedEventHandler objects are registered with the Framework service registry and are notified with an event object when an event is sent.

UntypedEventHandler objects must be registered with a service property TypedEventConstants.TYPED_EVENT_TOPICS whose value is the list of topics in which the event handler is interested.

For example:

```
String[] topics = new String[] {
  "com/isv/*"
};
Hashtable ht = new Hashtable();
ht.put(EventConstants.TYPE_SAFE_EVENT_TOPICS, topics);
context.registerService(UntypedEventHandler.class, this, ht);
```

Concurrency Thread-safe

157.9.7.1 public void notifyUntyped(String topic, Map<String, Object> event)

topic The topic to which the event was sent

event The event that occurred.

□ Called by the TypedEventBus service to notify the listener of an event.

157.10 org.osgi.service.typedevent.annotations

Typed Event Annotations Package Version 1.0.

This package contains annotations that can be used to require the Typed Event implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

157.10.1 Summary

- `RequireTypedEvent` - This annotation can be used to require the Typed Event implementation.

157.10.2 @RequireTypedEvent

This annotation can be used to require the Typed Event implementation. It can be used directly, or as a meta-annotation.

This annotation is applied to several of the Typed Event component property type annotations meaning that it does not normally need to be applied to Declarative Services components which use the Typed Event specification.

Since 1.0

Retention `CLASS`

Target `TYPE`, `PACKAGE`

157.11 org.osgi.service.typedevent.monitor

Typed Event Monitoring Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.typedevent.monitor; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.typedevent.monitor; version="[1.0,1.1)"`

157.11.1 Summary

- `MonitorEvent` - A monitoring event.
- `TypedEventMonitor` - The EventMonitor service can be used to monitor the events that are sent using the EventBus, and that are received from remote EventBus instances

157.11.2 public class MonitorEvent

A monitoring event.

Provider Type Consumers of this API must not implement this type

157.11.2.1 public Map<String, Object> eventData

The Data from the Event in Map form

157.11.2.2 public Instant publicationTime

The time at which the event was published

157.11.2.3 public String topic

The Event Topic

157.11.2.4 public MonitorEvent()

157.11.3 public interface TypedEventMonitor

The EventMonitor service can be used to monitor the events that are sent using the EventBus, and that are received from remote EventBus instances

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

157.11.3.1 public PushStream<MonitorEvent> monitorEvents()

□ Get a stream of events, starting now.

Returns A stream of event data

157.11.3.2 public PushStream<MonitorEvent> monitorEvents(int history)

history The requested number of historical events, note that fewer than this number of events may be returned if history is unavailable, or if insufficient events have been sent.

□ Get a stream of events, including up to the requested number of historical data events.

Returns A stream of event data

157.11.3.3 public PushStream<MonitorEvent> monitorEvents(Instant history)

history The requested time after which historical events, should be included. Note that events may have been discarded, or history unavailable.

□ Get a stream of events, including historical data events prior to the supplied time

Returns A stream of event data

157.12 org.osgi.service.typedevent.propertytypes

Typed Event Component Property Types Package Version 1.0.

When used as annotations, component property types are processed by tools to generate Component Descriptions which are used at runtime.

Bundles wishing to use this package at runtime must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.typedevent.propertytypes; version="[1.0,2.0)"`

157.12.1 Summary

- `EventFilter` - Component Property Type for the TypedEventConstants.TYPED_EVENT_FILTER service property of an Event Handler service.
- `EventTopics` - Component Property Type for the TypedEventConstants.TYPED_EVENT_TOPICS service property of a TypedEventHandler or UntypedEventHandler service.
- `EventType` - Component Property Type for the TypedEventConstants.TYPED_EVENT_TYPE service property of an TypedEventHandler service.

157.12.2 @EventFilter

Component Property Type for the TypedEventConstants.TYPED_EVENT_FILTER service property of an Event Handler service.

This annotation can be used on an TypedEventHandler or UntypedEventHandler component to declare the value of the TypedEventConstants.TYPED_EVENT_FILTER service property.

See Also Component Property Types

Retention CLASS

Target TYPE

157.12.2.1 String value

□ Service property specifying the event filter for a TypedEventHandler or UntypedEventHandler service.

Returns The event filter.

See Also TypedEventConstants.TYPED_EVENT_FILTER

157.12.3 @EventTopics

Component Property Type for the TypedEventConstants.TYPED_EVENT_TOPICS service property of a TypedEventHandler or UntypedEventHandler service.

This annotation can be used on a component to declare the values of the TypedEventConstants.TYPED_EVENT_TOPICS service property.

See Also Component Property Types

Retention CLASS

Target TYPE

157.12.3.1 String[] value

□ Service property specifying the Event topics of interest to an TypedEventHandler or UntypedEventHandler service.

Returns The event topics.

See Also TypedEventConstants.TYPED_EVENT_TOPICS

157.12.4 @EventType

Component Property Type for the TypedEventConstants.TYPED_EVENT_TYPE service property of an TypedEventHandler service.

This annotation can be used on an TypedEventHandler component to declare the value of the TypedEventConstants.TYPED_EVENT_TYPE service property.

See Also Component Property Types

Retention CLASS

Target TYPE

157.12.4.1 Class<?> value

□ Service property specifying the EventType for a TypedEventHandler service.

Returns The event filter.

See Also TypedEventConstants.TYPED_EVENT_TYPE

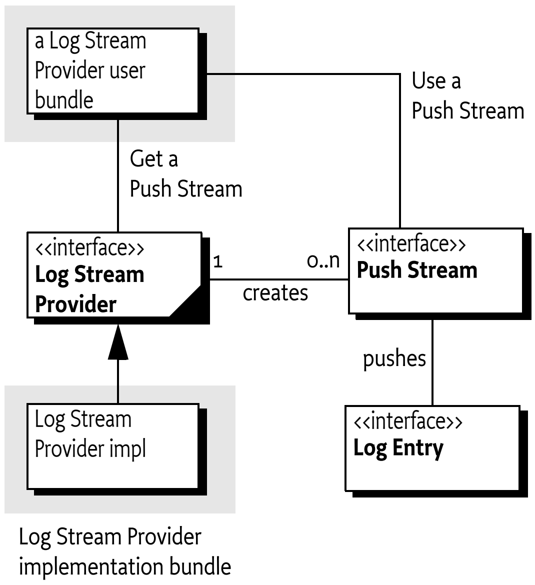
158 Log Stream Provider Service Specification

Version 1.0

158.1 Introduction

The Log Stream Provider service can be used to create Push Streams of Log Entries. Since the log is basically an ongoing stream of Log Entries having asynchronous arrival, a Push Stream of LogEntry objects can be used receive the Log Entries. See *Push Stream Specification* on page 1421 for information on Push Streams and how to use them.

Figure 158.1 Log Stream Diagram org.osgi.service.log.stream package


a Log Stream Provider user bundle
Use a Push Stream
Get a Push Stream
<<interface>> Log Stream Provider
1
0..n
creates
<<interface>> Push Stream
pushes
Log Stream Provider impl
<<interface>> Log Entry
Log Stream Provider implementation bundle

This specification defines the methods and semantics of interfaces which bundle developers can use to retrieve log entries.

Bundles can use the Log Stream Provider to retrieve Log Entry objects that were recorded recently or to receive Log Entry objects as they are logged by other bundles.

158.1.1 Entities

- *LogEntry* - An interface that allows access to a log entry in the log. It includes all the information that can be logged through the Logger as well as a time stamp, a sequence number, thread information, and location information. See [1] *Log Service* for more information about LogEntry.
- *LogStreamProvider* - A service interface that allows access to a PushStream of LogEntry objects.

158.2 Log Stream Provider

Push Streams created by the LogStreamProvider must:

- Be buffered with a buffer large enough to contain the history, if included.
- Have the QueuePolicyOption.DISCARD_OLDEST queue policy option.
- Use a shared executor.
- Have a parallelism of one.

The following code snippet show how one could get future Log Entries and print them.

```
logStreamProvider.createStream()
  .forEach(l -> System.out.println(l))
  .onResolve(() -> System.out.println("stream closed"));
```

The LogStreamProvider service offers a HISTORY option which will prime the returned Push Stream with the available log history, if any. The following code will process the available historical log entries followed by any new log entries.

```
logStreamProvider.createStream(LogStreamProvider.Options.HISTORY)
  .forEach(l -> System.out.println(l))
  .onResolve(() -> System.out.println("stream closed"));
```

158.3 Capabilities

The bundle providing the LogStreamProvider service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.log.stream package:

```
Provide-Capability: osgi.service;
 objectClass:List<String>="org.osgi.service.log.stream.LogStreamProvider";
 uses:="org.osgi.service.log.stream"
```

This capability must follow the rules defined for the osgi.service Namespace.

158.4 Security

The Log Stream Provide Service specification should only be implemented by trusted bundles. These bundles require ServicePermission[LogStreamProvider, REGISTER].

Only trusted bundles who must be able to access log entries should be assigned ServicePermission[LogStreamProvider, GET].

158.5 org.osgi.service.log.stream

Log Stream Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.log.stream; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.log.stream; version="[1.0,1.1)"
```

158.5.1 Summary

- `LogStreamProvider` - LogStreamProvider service for creating a PushStream of LogEntry objects.
- `LogStreamProvider.Options` - Creation options for the PushStream of LogEntry objects.

158.5.2 public interface LogStreamProvider

LogStreamProvider service for creating a PushStream of LogEntry objects.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

158.5.2.1 public PushStream<LogEntry> createStream(LogStreamProvider.Options... options)

options The options to use when creating the PushStream.

□ Create a PushStream of LogEntry objects.

The returned PushStream must:

- Be buffered with a buffer large enough to contain the history, if included.
- Have the QueuePolicyOption.DISCARD_OLDEST queue policy option.
- Use a shared executor.
- Have a parallelism of one.

When this LogStreamProvider service is released by the obtaining bundle, this LogStreamProvider service must call PushStream.close() on the returned PushStream object if it has not already been closed.

Returns A PushStream of LogEntry objects.

158.5.3 enum LogStreamProvider.Options

Creation options for the PushStream of LogEntry objects.

158.5.3.1 HISTORY

Include history.

Prime the created PushStream with the available historical LogEntry objects. The number of available LogEntry objects is implementation specific.

The created PushStream will supply the available historical LogEntry objects followed by newly created LogEntry objects.

158.5.3.2 public static LogStreamProvider.Options valueOf(String name)

158.5.3.3 public static LogStreamProvider.Options[] values()

158.6 References

[1] *Log Service*

OSGi Core, Chapter 101 Log Service Specification

159 Feature Service Specification

Version 1.0

159.1 Introduction

OSGi has become a platform capable of running large applications for a variety of purposes, including rich client applications, server-side systems and cloud and container based architectures. As these applications are generally based on many bundles, describing each bundle individually in an application definition becomes unwieldy once the number of bundles reaches a certain level.

When developing large scale applications it is often the case that few people know the role of every single bundle or configuration item in the application. To keep the architecture understandable a grouping mechanism is needed that allows for the representation of parts of the application into larger entities that keep reasoning about the system manageable. In such a domain members of teams spread across an organization will need to be able to both develop new parts for the application as well as make tweaks or enhancements to parts developed by others such as adding configuration and resources or changing one or more bundles relevant to their part of the application.

The higher level constructs that define the application should be reusable in different contexts, for example if one team has developed a component to handle job processing, different applications should be able to use it, and if needed tune its configuration or other aspects so that it works in each setting without having to know each and every detail that the job processing component is built up from.

Applications are often associated with additional resources or metadata, for example database scripts or custom artifacts. By including these with the application definition, all the related entities are encapsulated in a single artifact.

By combining various applications or subsystems together, systems are composed of existing, reusable building blocks, where all these blocks can work together. Architects of these systems need to think about components without having to dive into the individual implementation details of each subcomponent. The Features defined in this specification can be used to model such applications. Features contain the definition of an application or component and may be composed into larger systems.

159.1.1 Essentials

- *Declarative* - Features are declarative and can be mapped to different implementations.
- *Extensible* - Features are extensible with custom content to facilitate all information related to a Feature to be co-located.
- *Human Readable* - No special software is needed to read or author Features.
- *Machine Readable* - Features are easily be processed by tools.

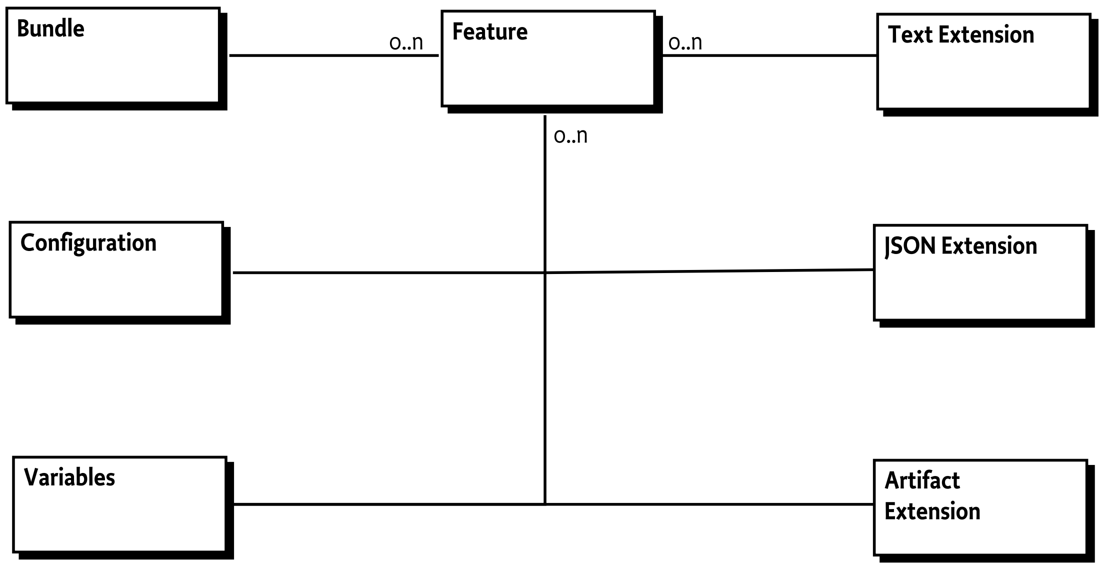
159.1.2 Entities

The following entities are used in this specification:

- *Feature* - A Feature contains a number of entities that, when provided to a launcher can be turned into an executable system. Features are building blocks which may be assembled into larger systems.

- *Bundles* - A Feature can contain one ore more bundles.
- *Configuration* - A Feature can contain configurations for the Configuration Admin service.
- *Extension* - A Feature can contain a number of extensions with custom content.
- *Launcher* - A launcher turns one or more Features into an executable system.
- *Processor* - A Feature processor reads Features and perform a processing operation on them, such as validation, transformation or generation of new entities based on the Features.
- *Properties* - Framework launching properties can be specified in a Feature.

Figure 159.1 Features Entity overview



159.2 Feature

Features are defined by declaring JSON documents or by using the Feature API. Each Feature has a unique ID which includes a version. It holds a number of entities, including a list of bundles, configurations and others. Features are extensible, that is a Feature can also contain any number of custom entities which are related to the Feature.

Features may have dependencies on other Features. Features inherit the capabilities and requirements from all bundles listed in the Feature.

Once created, a Feature is immutable. Its definition cannot be modified. However it is possible to record caching related information in a Feature through transient extensions. This cached content is not significant for the definition of the Feature or part of its identity.

159.2.1 Identifiers

Identifiers used throughout this specification are defined using the Maven Identifier model. They are composed of the following parts:

- Group ID
- Artifact ID
- Version
- Type (optional)

- Classifier (optional)

Note that if Version has the -SNAPSHOT suffix, the identifier points at an unreleased artifact that is under development and may still change.

For more information see [3] *Apache Maven Pom Reference*. The format used to specify identifiers is as follows:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

159.2.2 Feature Identifier

Each Feature has a unique identifier. Apart from providing a persistent handle to the Feature, it also provides enough information to find the Feature in an artifact repository. This identifier is defined using the format described in *Identifiers* on page 1358.

159.2.2.1 Identifier type

Features use as identifier type the value osgifeature.

159.2.3 Attributes

A Feature can have the following attributes:

Table 159.1 Feature Attributes

Attribute	Data Type	Kind	Description
name	String	Optional	The short descriptive name of the Feature.
categories	Array of String	Optional, defaults to an empty array	The categories this Feature belongs to. The values are user-defined.
complete	boolean	Optional, defaults to false	Completeness of the Feature. A Feature is complete when it has no external dependencies.
description	String	Optional	A longer description of the Feature.
docURL	String	Optional	A location where documentation can be found for the Feature.
license	String	Optional	The license of the Feature. The license only relates to the Feature itself and not to any artifacts that might be referenced by the Feature. The license follows the Bundle-License format as specified in the Core specification.
SCM	String	Optional	SCM information relating to the feature. The syntax of the value follows the Bundle-SCM format. See the 'Bundle Manifest Headers' section in the OSGi Core specification.
vendor	String	Optional	The vendor of the Feature.

An initial Feature without content can be declared as follows:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.0",

  "name": "The ACME app",
  "description":
    "This is the main ACME app, from where all functionality is reached."

  /*
    Additional Feature entities here
    ...
  */
}
```

159.2.4 Using the Feature API

Features can also be created, read and written using the Feature API. The main entry point for this API is the FeatureService. The Feature API uses the builder pattern to create entities used in Features.

A builder instance is used to create a single entity and cannot be re-used to create a second one. Builders are created from the BuilderFactory, which is available from the FeatureService through getBuilderFactory().

```
FeatureService fs = ... // from Service Registry
BuilderFactory factory = fs.getBuilderFactory();

FeatureBuilder builder = factory.newFeatureBuilder(
  fs.getID("org.acme", "acmeapp", "1.0.0"));
builder.setName("The ACME app");
builder.setDescription("This is the main ACME app, "
  + "from where all functionality is reached.");

Feature f = builder.build();
```

The Feature API can also be useful in environments outside of an OSGi Framework where no service registry is available, for example in a build-system environment. In such environments the FeatureService can be obtained by using the java.util.ServiceLoader mechanism.

159.3 Comments

Comments in the form of [2] *JSMin (The JavaScript Minifier)* comments are supported, that is, any text on the same line after // is ignored and any text between /* */ is ignored.

159.4 Bundles

Features list zero or more bundles that implement the functionality provided by the Feature. Bundles are listed by referencing them in the bundles array so that they can be resolved from a repository. Bundles can have metadata associated with them, such as the relative start order of the bundle in the Feature. Custom metadata may also be provided. A single Feature can provide multiple versions of the same bundle, if desired.

Bundles are referenced using the identifier format described in *Identifiers* on page 1358. This means that Bundles are referenced using their Maven coordinates. The bundles array contains JSON objects which can contain the bundle IDs and specify optional additional metadata.

159.4.1 Bundle Metadata

Arbitrary key-value pairs can be associated with bundle entries to store custom metadata alongside the bundle references. Reverse DNS naming should be used with the keys to avoid name clashes when metadata is provided by multiple entities. Keys not using the reverse DNS naming scheme are reserved for OSGi use.

Bundle metadata supports string keys and string, number or boolean values.

The following example shows a simple Feature describing a small application with its dependencies:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.1",

  "name": "The Acme Application",
  "license": "https://opensource.org/licenses/Apache-2.0",
  "complete": true,

  "bundles": [
    { "id": "org.osgi:org.osgi.util.function:1.1.0" },
    { "id": "org.osgi:org.osgi.util.promise:1.1.1" },
    {
      "id": "org.apache.commons:commons-email:1.5",

      // This attribute is used by custom tooling to
      // find the associated javadoc
      "org.acme.javadoc.link":
        "https://commons.apache.org/proper/commons-email/javadocs/api-1.5"
    },
    { "id": "com.acme:acmelib:1.7.2" }
  ]

  /*
    Additional Feature entities here
    ...
  */
}
```

159.4.2 Using the Feature API

A Feature with Bundles can be created using the Feature API as follows:

```
FeatureService fs = ... // from Service Registry
BuilderFactory factory = fs.getBuilderFactory();

FeatureBuilder builder = factory.newFeatureBuilder(
  fs.getID("org.acme", "acmeapp", "1.0.1"));
builder.setName("The Acme Application");
builder.setLicense("https://opensource.org/licenses/Apache-2.0");
builder.setComplete(true);

FeatureBundle b1 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.osgi:org.osgi.util.function:1.1.0"))
  .build();
FeatureBundle b2 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.osgi:org.osgi.util.promise:1.1.1"))
  .build();
```

```
FeatureBundle b3 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.apache.commons:commons-email:1.1.5"))
  .addMetadata("org.acme.javadoc.link",
    "https://commons.apache.org/proper/commons-email/javadocs/api-1.5")
  .build();
FeatureBundle b4 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "com.acme:acmelib:1.7.2"))
  .build();

builder.addBundles(b1, b2, b3, b4);
Feature f = builder.build();
```

159.5 Configurations

Features support configuration using the OSGi Configurator syntax, see *Configurator Specification* on page 1145. This is specified with the configurations key in the Feature. A Launcher can apply these configurations to the Configuration Admin service when starting the system.

It is an error to define the same PID twice in a single Feature. An entity processing the feature must fail in this case.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:osgifeature:configs:1.0.0",
    "configurations": {
        "org.apache.felix.http": {
            "org.osgi.service.http.port": 8080,
            "org.osgi.service.http.port.secure": 8443
        }
    }
}
```

159.6 Variables

Configurations and Framework Launching Properties support late binding of values. This enables setting these items through a Launcher, for example to specify a database user name, server port number or other information that may be variable between runtimes.

Variables are declared in the variables section of the Feature and they can have a default value specified. The default must be of type string, number or boolean. Variables can also be declared to *not* have a default, which means that they must be provided with a value through the Launcher. This is done by specifying null as the default in the variable declaration.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:osgifeature:configs:1.1.0",
    "variables": {
        "http.port": 8080,
        "db.username": "scott",
        "db.password": null
    },
    "configurations": {
        "org.acme.server.http": {
            "org.osgi.service.http.port:Integer": "${http.port}"
        },
        "org.acme.db": {
            "username": "${db.username}-user",
            "password": "${db.password}"
        }
    }
}
```

Variables are referenced with the curly brace placeholder syntax: `${` *`variable-name`* `}` in the configuration value or framework launching property value section. To support conversion of variables to non-string types the configurator syntax specifying the datatype with the configuration key is used, as in the above example.

Multiple variables can be referenced for a single configuration or framework launching property value and variables may be combined with text. If no variable exist with the given name, then the `${` *`variable-name`* `}` must be retained in the value.

159.7 Extensions

Features can include custom content. This makes it possible to keep custom entities and information relating to the Feature together with the rest of the Feature.

Custom content is provided through Feature extensions, which are in one of the following formats:

- *Text* - A text extension contains an array of text.
- *JSON* - A JSON extension contains embedded custom JSON content.
- *Artifacts* - A list of custom artifacts associated with the Feature.

Extensions can have a variety of consumers. For example they may be handled by a Feature Launcher or by an external tool which can process the extension at any point of the Feature life cycle.

Extensions are of one of the following three kinds:

- *Mandatory* - The entity processing this Feature *must* know how to handle this extension. If it cannot handle the extension it must fail.
- *Optional* - This extension is optional. If the entity processing the Feature cannot handle it, the extension can be skipped or ignored. This is the default.
- *Transient* - This extension contains transient information which may be used to optimize the processing of the Feature. It is not part of the Feature definition.

Extensions are specified as JSON objects under the `extensions` key in the Feature. A Feature can contain any number of extensions, as long as the extension keys are unique. Extension keys should use reverse domain naming to avoid name clashing of multiple extensions in a single Feature. Extensions names without a reverse domain naming prefix are reserved for OSGi use.

159.7.1 Text Extensions

Text extensions support the addition of custom text content to the Feature. The text is provided as a JSON array of strings.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.0.0",

    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",

    "extensions": {
        "org.acme.mydoc": {
            "type": "text",
            "text": [
                "This application provides the main acme ",
                "functionality."
            ]
        }
    }
}
```

159.7.2 JSON Extensions

Custom JSON content is added to Features by using a JSON extension. The content can either be a JSON object or a JSON array.

The following example extension declares under which execution environment the Feature is complete, using a custom JSON object.

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.1.0",

    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",

    "extensions": {
        "org.acme.execution-environment": {
            "type": "json",
            "json": {
                "environment-capabilities":
                    ["osgi.ee; filter:=\"(&(osgi.ee=JavaSE)(version=11))\""],
                "framework": "org.osgi:core:6.0.0",
                "provided-features": ["org.acme:platform:1.1"]
            }
        }
    }
}
```

159.7.3 Artifact list Extensions

Custom extensions can be used to associate artifacts that are not listed as bundles with the Feature.

For example, database definition resources may be listed as artifacts in a Feature. In the following example, the extension org.acme.ddlfiles lists Database Definition Resources which *must* be handled by the launcher agent, that is, the database must be configured when the application is run:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.2.0",
    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",
    "complete": true,

    "bundles": [
        "org.osgi:org.osgi.util.function:1.1.0",
        "org.osgi:org.osgi.util.promise:1.1.1",
        "com.acme:acmelib:2.0.0"
    ],

    "extensions": {
        "org.acme.ddlfiles": {
            "kind": "mandatory",
            "type": "artifacts",
            "artifacts": [
              { "id": "org.acme:appddl:1.2.1" },
              {
                "id": "org.acme:appddl-custom:1.0.3",
                "org.acme.target": "custom-db"
              }
            ]
        }
    }
}
```

As with bundle identifiers, custom artifacts are specified in an object in the artifacts list with an explicit id and optional additional metadata. The keys of the metadata should use a reverse domain naming pattern to avoid clashes. Keys that do not use reverse domain name as a prefix are reserved for OSGi use. Supported metadata values must be of type string, number or boolean.

159.8 Framework Launching Properties

When a Feature is launched in an OSGi framework it may be necessary to specify Framework Properties. These are provided in the Framework Launching Properties extension section of the Feature. The Launcher must be able to satisfy the specified properties. If it cannot ensure that these are present in the running Framework the launcher must fail.

Framework Launching Properties can reference Variables as defined in *Variables* on page 1362. These variables are substituted before the properties are set.

Example:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:osgifeature:fw-props:2.0.0",

  "variables": {
    "fw.storage.dir": "/tmp" // Can be overridden through the launcher
  },

  "extensions": {
    "framework-launching-properties": {
      "type": "json",
      "json": {
          "org.osgi.framework.system.packages.extra":
            "javax.activation;version=\"1.1.1\"",
          "org.osgi.framework.bootdelegation": "javax.activation",
          "org.osgi.framework.storage": "${fw.storage.dir}"
      }
    }
  }
}
```

159.9 Resource Versioning

Feature JSON resources are versioned to support updates to the JSON structure in the future. To declare the document version of the Feature use the feature-resource-version key in the JSON document.

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.0"

  /*
    Additional Feature entities here
    ...
  */
}
```

The currently supported version of the Feature JSON documents is 1.0. If no Feature Resource Version is specified 1.0 is used as the default.

159.10 Capabilities

159.10.1 osgi.service Capability

The bundle providing the Feature Service must provide a capability in the osgi.service namespace representing the services it is registering. This capability must also declare uses constraints for the relevant service packages:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.feature.FeatureService";
  uses:="org.osgi.service.feature"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 711.

159.11 org.osgi.service.feature

Feature Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.feature; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.feature; version="[1.0,1.1)"
```

159.11.1 Summary

- `BuilderFactory` - The Builder Factory can be used to obtain builders for the various entities.
- `Feature` - The Feature Model Feature.
- `FeatureArtifact` - An Artifact is an entity with an ID, for use in extensions.
- `FeatureArtifactBuilder` - A builder for FeatureArtifact objects.
- `FeatureBuilder` - A builder for Feature Models.
- `FeatureBundle` - A Bundle which is part of a feature.
- `FeatureBundleBuilder` - A builder for Feature Model FeatureBundle objects.
- `FeatureConfiguration` - Represents an OSGi Configuration in the Feature Model.
- `FeatureConfigurationBuilder` - A builder for Feature Model FeatureConfiguration objects.
- `FeatureConstants` - Defines standard constants for the Feature specification.
- `FeatureExtension` - A Feature Model Extension.
- `FeatureExtension.Kind` - The kind of extension: optional, mandatory or transient.
- `FeatureExtension.Type` - The type of extension
- `FeatureExtensionBuilder` - A builder for Feature Model FeatureExtension objects.
- `FeatureService` - The Feature service is the primary entry point for interacting with the feature model.
- `ID` - ID used to denote an artifact.

159.11.2 public interface BuilderFactory

The Builder Factory can be used to obtain builders for the various entities.

Provider Type Consumers of this API must not implement this type

159.11.2.1 public FeatureArtifactBuilder newArtifactBuilder(ID id)

id The artifact ID for the artifact object being built.

□ Obtain a new builder for Artifact objects.

Returns The builder.

159.11.2.2 public FeatureBundleBuilder newBundleBuilder(ID id)

id The ID for the bundle object being built. If the ID has no type specified, a default type of @{code jar} is assumed.

□ Obtain a new builder for Bundle objects.

Returns The builder.

159.11.2.3 public FeatureConfigurationBuilder newConfigurationBuilder(String pid)

pid The persistent ID for the Configuration being built.

□ Obtain a new builder for Configuration objects.

Returns The builder.

159.11.2.4 public FeatureConfigurationBuilder newConfigurationBuilder(String factoryPid, String name)

factoryPid The factory persistent ID for the Configuration being built.

name The name of the configuration being built. The PID for the configuration will be the factoryPid + '~' + name

□ Obtain a new builder for Factory Configuration objects.

Returns The builder.

159.11.2.5 public FeatureExtensionBuilder newExtensionBuilder(String name, FeatureExtension.Type type, FeatureExtension.Kind kind)

name The extension name.

type The type of extension: JSON, Text or Artifacts.

kind The kind of extension: Mandatory, Optional or Transient.

□ Obtain a new builder for Feature objects.

Returns The builder.

159.11.2.6 public FeatureBuilder newFeatureBuilder(ID id)

id The ID for the feature object being built. If the ID has no `type` specified, a default type of `osgifeature` is assumed.

□ Obtain a new builder for Feature objects.

Returns The builder.

159.11.3 public interface Feature

The Feature Model Feature.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.3.1 public List<FeatureBundle> getBundles()

□ Get the bundles.

Returns The bundles. The returned list is unmodifiable.

159.11.3.2 public List<String> getCategories()

□ Get the categories.

Returns The categories. The returned list is unmodifiable.

159.11.3.3 public Map<String, FeatureConfiguration> getConfigurations()

□ Get the configurations. The iteration order of the returned map should follow the definition order of the configurations in the feature.

Returns The configurations. The returned map is unmodifiable.

159.11.3.4 public Optional<String> getDescription()

□ Get the description.

Returns The description.

159.11.3.5 public Optional<String> getDocURL()

□ Get the documentation URL.

Returns The documentation URL.

159.11.3.6 public Map<String, FeatureExtension> getExtensions()

□ Get the extensions. The iteration order of the returned map should follow the definition order of the extensions in the feature.

Returns The extensions. The returned map is unmodifiable.

159.11.3.7 public ID getID()

□ Get the Feature's ID.

Returns The ID of this Feature.

159.11.3.8 public Optional<String> getLicense()

□ Get the license of this Feature. The syntax of the value follows the Bundle-License header syntax. See the 'Bundle Manifest Headers' section in the OSGi Core specification.

Returns The license.

159.11.3.9 public Optional<String> getName()

□ Get the name.

Returns The name.

159.11.3.10 public Optional<String> getSCM()

□ Get the SCM information relating to the feature. The syntax of the value follows the Bundle-SCM format. See the 'Bundle Manifest Headers' section in the OSGi Core specification.

Returns The SCM information.

159.11.3.11 public Map<String, Object> getVariables()

□ Get the variables. The iteration order of the returned map should follow the definition order of the variables in the feature. Values are of type: String, Boolean or BigDecimal for numbers. The null JSON value is represented by a null value in the map.

Returns The variables. The returned map is unmodifiable.

159.11.3.12 public Optional<String> getVendor()

□ Get the vendor.

Returns The vendor.

159.11.3.13 public boolean isComplete()

□ Get whether the feature is complete or not.

Returns Completeness value.

159.11.4 public interface FeatureArtifact

An Artifact is an entity with an ID, for use in extensions.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.4.1 public ID getID()

□ Get the artifact's ID.

Returns The ID of this artifact.

159.11.4.2 public Map<String, Object> getMetadata()

□ Get the metadata for this artifact.

Returns The metadata. The returned map is unmodifiable.

159.11.5 public interface FeatureArtifactBuilder

A builder for FeatureArtifact objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.5.1 public FeatureArtifactBuilder addMetadata(String key, Object value)

key Metadata key.

value Metadata value.

□ Add metadata for this Artifact.

Returns This builder.

159.11.5.2 public FeatureArtifactBuilder addMetadata(Map<String, Object> metadata)

metadata The map with metadata.

□ Add metadata for this Artifact by providing a map. All metadata in the map is added to any previously provided metadata.

Returns This builder.

159.11.5.3 public FeatureArtifact build()

□ Build the Artifact object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Feature Artifact.

159.11.6 public interface FeatureBuilder

A builder for Feature Models.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.6.1 public FeatureBuilder addBundles(FeatureBundle... bundles)

bundles The Bundles to add.

□ Add Bundles to the Feature.

Returns This builder.

159.11.6.2 public FeatureBuilder addCategories(String... categories)

categories The Categories.

□ Adds one or more categories to the Feature.

Returns This builder.

159.11.6.3 public FeatureBuilder addConfigurations(FeatureConfiguration... configs)

configs The Configurations to add.

□ Add Configurations to the Feature.

Returns This builder.

159.11.6.4 public FeatureBuilder addExtensions(FeatureExtension... extensions)

extensions The Extensions to add.

□ Add Extensions to the Feature

Returns This builder.

159.11.6.5 public FeatureBuilder addVariable(String key, Object defaultValue)

key The key.

defaultValue The default value.

□ Add a variable to the Feature. If a variable with the specified key already exists it is replaced with this one. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

159.11.6.6 public FeatureBuilder addVariables(Map<String, Object> variables)

variables to be added.

□ Add a map of variables to the Feature. Pre-existing variables with the same key in are overwritten if these keys exist in the map. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type.

159.11.6.7 public Feature build()

□ Build the Feature. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Feature.

159.11.6.8 public FeatureBuilder setComplete(boolean complete)

complete If the feature is complete.

□ Set the Feature Complete flag. If this method is not called the complete flag defaults to false.

Returns This builder.

159.11.6.9 public FeatureBuilder setDescription(String description)

description The description.

□ Set the Feature Description.

Returns This builder.

159.11.6.10 public FeatureBuilder setDocURL(String docURL)

docURL The Documentation URL.

□ Set the documentation URL.

Returns This builder.

159.11.6.11 public FeatureBuilder setLicense(String license)

license The License.

□ Set the License.

Returns This builder.

159.11.6.12 public FeatureBuilder setName(String name)

name The Name.

□ Set the Feature Name.

Returns This builder.

159.11.6.13 public FeatureBuilder setSCM(String scm)

scm The SCM information.

□ Set the SCM information.

Returns This builder.

159.11.6.14 public FeatureBuilder setVendor(String vendor)

vendor The Vendor.

□ Set the Vendor.

Returns This builder.

159.11.7 public interface FeatureBundle

A Bundle which is part of a feature.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.7.1 public ID getID()

□ Get the bundle's ID.

Returns The ID of this bundle.

159.11.7.2 public Map<String, Object> getMetadata()

□ Get the metadata for this bundle.

Returns The metadata. The returned map is unmodifiable.

159.11.8 public interface FeatureBundleBuilder

A builder for Feature Model FeatureBundle objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.8.1 public FeatureBundleBuilder addMetadata(String key, Object value)

key Metadata key.

value Metadata value.

□ Add metadata for this Bundle.

Returns This builder.

159.11.8.2 public FeatureBundleBuilder addMetadata(Map<String, Object> metadata)

metadata The map with metadata.

□ Add metadata for this Bundle by providing a map. All metadata in the map is added to any previously provided metadata.

Returns This builder.

159.11.8.3 public FeatureBundle build()

□ Build the Bundle object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Bundle.

159.11.9 public interface FeatureConfiguration

Represents an OSGi Configuration in the Feature Model.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.9.1 public Optional<String> getFactoryPid()

□ Get the Factory PID from the configuration, if any.

Returns The Factory PID, or null if there is none.

159.11.9.2 public String getPid()

□ Get the PID from the configuration.

Returns The PID.

159.11.9.3 public Map<String, Object> getValues()

□ Get the configuration key-value map.

Returns The key-value map. The returned map is unmodifiable.

159.11.10 public interface FeatureConfigurationBuilder

A builder for Feature Model FeatureConfiguration objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.10.1 public FeatureConfigurationBuilder addValue(String key, Object value)

key The configuration key.

value The configuration value. Acceptable data types are the data type supported by the Configuration Admin service, which are the Primary Property Types as defined for the Filter Syntax in the OSGi Core specification.

□ Add a configuration value for this Configuration object. If a value with the same key was previously provided (regardless of case) the previous value is overwritten.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

159.11.10.2 public FeatureConfigurationBuilder addValues(Map<String, Object> configValues)

configValues The map of configuration values to add. Acceptable value types are the data type supported by the Configuration Admin service, which are the Primary Property Types as defined for the Filter Syntax in the OSGi Core specification.

□ Add a map of configuration values for this Configuration object. Values will be added to any previously provided configuration values. If a value with the same key was previously provided (regardless of case) the previous value is overwritten.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type or if the same key is provided in different capitalizations (regardless of case).

159.11.10.3 public FeatureConfiguration build()

□ Build the Configuration object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Configuration.

159.11.11 public final class FeatureConstants

Defines standard constants for the Feature specification.

159.11.11.1 public static final String FEATURE_IMPLEMENTATION = "osgi.feature"

The name of the implementation capability for the Feature specification.

159.11.11.2 public static final String FEATURE_SPECIFICATION_VERSION = "1.0"

The version of the implementation capability for the Feature specification.

159.11.12 public interface FeatureExtension

A Feature Model Extension. Extensions can contain either Text, JSON or a list of Artifacts.

Extensions are of one of the following kinds:

- Mandatory: this extension must be processed by the runtime
- Optional: this extension does not have to be processed by the runtime
- Transient: this extension contains transient information such as caching data that is for optimization purposes. It may be changed or removed and is not part of the feature's identity.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.12.1 public List<FeatureArtifact> getArtifacts()

□ Get the Artifacts from this extension.

Returns The Artifacts. The returned list is unmodifiable.

Throws `IllegalStateException`– If called on an extension which is not of type ARTIFACTS.

159.11.12.2 public String getJSON()

□ Get the JSON from this extension.

Returns The JSON.

Throws `IllegalStateException`– If called on an extension which is not of type JSON.

159.11.12.3 public FeatureExtension.Kind getKind()

□ Get the extension kind.

Returns The kind.

159.11.12.4 public String getName()

□ Get the extension name.

Returns The name.

159.11.12.5 public List<String> getText()

□ Get the Text from this extension.

Returns The lines of text. The returned list is unmodifiable.

Throws `IllegalStateException`– If called on an extension which is not of type TEXT.

159.11.12.6 public FeatureExtension.Type getType()

□ Get the extension type.

Returns The type.

159.11.13 enum FeatureExtension.Kind

The kind of extension: optional, mandatory or transient.

159.11.13.1 MANDATORY

A mandatory extension must be processed.

159.11.13.2 OPTIONAL

An optional extension can be ignored if no processor is found.

159.11.13.3 TRANSIENT

A transient extension contains computed information which can be used as a cache to speed up operation.

159.11.13.4 public static FeatureExtension.Kind valueOf(String name)

159.11.13.5 public static FeatureExtension.Kind[] values()

159.11.14 enum FeatureExtension.Type

The type of extension

159.11.14.1 JSON

A JSON extension.

159.11.14.2 TEXT

A plain text extension.

159.11.14.3 ARTIFACTS

An extension that is a list of artifact identifiers.

159.11.14.4 public static FeatureExtension.Type valueOf(String name)

159.11.14.5 public static FeatureExtension.Type[] values()

159.11.15 public interface FeatureExtensionBuilder

A builder for Feature Model FeatureExtension objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.15.1 public FeatureExtensionBuilder addArtifact(FeatureArtifact artifact)

artifact The artifact to add.

□ Add an Artifact to the extension. Can only be called for extensions of type FeatureExtension.Type.ARTIFACTS.

Returns This builder.

159.11.15.2 public FeatureExtensionBuilder addText(String text)

text The text to be added.

□ Add a line of text to the extension. Can only be called for extensions of type FeatureExtension.Type.TEXT.

Returns This builder.

159.11.15.3 public FeatureExtension build()

□ Build the Extension. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Extension.

159.11.15.4 public FeatureExtensionBuilder setJSON(String json)

json The JSON to be added.

□ Add JSON in String form to the extension. Can only be called for extensions of type FeatureExtension.Type.JSON.

Returns This builder.

159.11.16 public interface FeatureService

The Feature service is the primary entry point for interacting with the feature model.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.16.1 public BuilderFactory getBuilderFactory()

□ Get a factory which can be used to build feature model entities.

Returns A builder factory.

159.11.16.2 public ID getID(String groupId, String artifactId, String version)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.3 public ID getID(String groupId, String artifactId, String version, String type)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

type The type (not null, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.4 public ID getID(String groupId, String artifactId, String version, String type, String classifier)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

type The type (not null, not empty).

classifier The classifier (not `null`, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.5 public ID getIDfromMavenCoordinates(String coordinates)

coordinates The Maven Coordinates.

□ Obtain an ID from a Maven Coordinates formatted string. The supported syntax is as follows:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

Returns the ID.

159.11.16.6 public Feature readFeature(Reader jsonReader) throws IOException

jsonReader A Reader to the JSON input

□ Read a Feature from JSON

Returns The Feature represented by the JSON

Throws `IOException`– When reading fails

159.11.16.7 public void writeFeature(Feature feature, Writer jsonWriter) throws IOException

feature the Feature to write.

jsonWriter A Writer to which the Feature should be written.

□ Write a Feature Model to JSON

Throws `IOException`– When writing fails.

159.11.17 public interface ID

ID used to denote an artifact. This could be a feature model, a bundle which is part of the feature model or some other artifact.

Artifact IDs follow the Maven convention of having:

- A group ID
- An artifact ID
- A version
- A type identifier (optional)
- A classifier (optional)

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.17.1 public static final String FEATURE_ID_TYPE = "osgifeature"

ID type for use with Features.

159.11.17.2 public String getArtifactId()

□ Get the artifact ID.

Returns The artifact ID.

159.11.17.3 public Optional<String> getClassifier()

□ Get the classifier.

Returns The classifier.

159.11.17.4 public String getGroupId()

□ Get the group ID.

Returns The group ID.

159.11.17.5 public Optional<String> getType()

□ Get the type identifier.

Returns The type identifier.

159.11.17.6 public String getVersion()

□ Get the version.

Returns The version.

159.11.17.7 public String toString()

□ This method returns the ID using the following syntax:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

Returns The string representation.

159.12 org.osgi.service.feature.annotation

Feature Annotations Package Version 1.0.

This package contains annotations that can be used to require the Feature Service implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

159.12.1 Summary

- `RequireFeatureService` - This annotation can be used to require the Feature implementation.

159.12.2 @RequireFeatureService

This annotation can be used to require the Feature implementation. It can be used directly, or as a meta-annotation.

Retention `CLASS`

Target `TYPE`, `PACKAGE`

159.13 References

[1] *JSON (JavaScript Object Notation)*
https://www.json.org

[2] *JSMin (The JavaScript Minifier)*
https://www.crockford.com/javascript/jsmin.html

[3] *Apache Maven Pom Reference*
https://maven.apache.org/pom.html

702 XML Parser Service Specification

Version 1.0

702.1 Introduction

The Extensible Markup Language (XML) has become a popular method of describing data. As more bundles use XML to describe their data, a common XML Parser becomes necessary in an embedded environment in order to reduce the need for space. Not all XML Parsers are equivalent in function, however, and not all bundles have the same requirements on an XML parser.

This problem was addressed in the Java API for XML Processing, see [4] *JAXP* for Java 2 Standard Edition and Enterprise Edition. This specification addresses how the classes defined in JAXP can be used in an OSGi framework. It defines how:

- Implementations of XML parsers can become available to other bundles
- Bundles can find a suitable parser
- A standard parser in a JAR can be transformed to a bundle

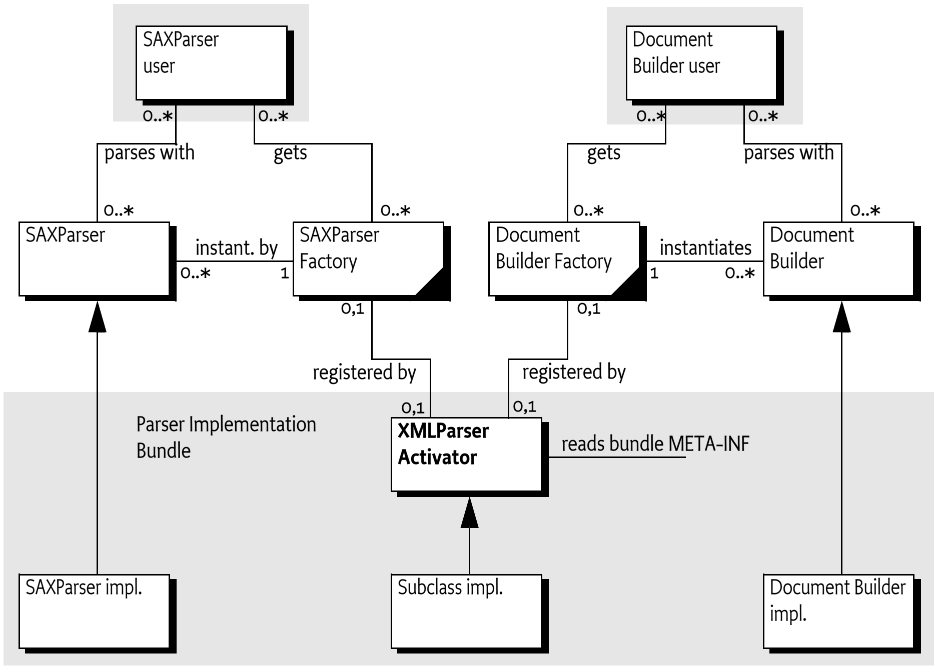
702.1.1 Essentials

- *Standards* - Leverage existing standards in Java based XML parsing: JAXP, SAX and DOM
- *Unmodified JAXP code* - Run unmodified JAXP code
- *Simple* - It should be easy to provide a SAX or DOM parser as well as easy to find a matching parser
- *Multiple* - It should be possible to have multiple implementations of parsers available
- *Extendable* - It is likely that parsers will be extended in the future with more functionality

702.1.2 Entities

- *XMLParserActivator* - A utility class that registers a parser factory from declarative information in the Manifest file.
- *SAXParserFactory* - A class that can create an instance of a `SAXParser` class.
- *DocumentBuilderFactory* - A class that can create an instance of a `DocumentBuilder` class.
- *SAXParser* - A parser, instantiated by a `SaxParserFactory` object, that parses according to the SAX specifications.
- *DocumentBuilder* - A parser, instantiated by a `DocumentBuilderFactory`, that parses according to the DOM specifications.

Figure 702.1 XML Parsing diagram



702.1.3 Operations

A bundle containing a SAX or DOM parser is started. This bundle registers a SAXParserFactory and/ or a DocumentBuilderFactory service object with the Framework. Service registration properties describe the features of the parsers to other bundles. A bundle that needs an XML parser will get a SAXParserFactory or DocumentBuilderFactory service object from the Framework service registry. This object is then used to instantiate the requested parsers according to their specifications.

702.2 JAXP

XML has become very popular in the last few years because it allows the interchange of complex information between different parties. Though only a single XML standard exists, there are multiple APIs to XML parsers, primarily of two types:

- The Simple API for XML (SAX1 and SAX2)
- Based on the Document Object Model (DOM 1 and 2)

Both standards, however, define an abstract API that can be implemented by different vendors.

A given XML Parser implementation may support either or both of these parser types by implementing the org.w3c.dom and/or org.xml.sax packages. In addition, parsers have characteristics such as whether they are validating or non-validating parsers and whether or not they are namespace aware.

An application which uses a specific XML Parser must code to that specific parser and become coupled to that specific implementation. If the parser has implemented [4] *JAXP*, however, the application developer can code against SAX or DOM and let the runtime environment decide which parser implementation is used.

JAXP uses the concept of a *factory*. A factory object is an object that abstracts the creation of another object. JAXP defines a DocumentBuilderFactory and a SAXParserFactory class for this purpose.

JAXP is implemented in the javax.xml.parsers package and provides an abstraction layer between an application and a specific XML Parser implementation. Using JAXP, applications can choose to use any JAXP compliant parser without changing any code, simply by changing a System property which specifies the SAX- and DOM factory class names.

In JAXP, the default factory is obtained with a static method in the SAXParserFactory or DocumentBuilderFactory class. This method will inspect the associated System property and create a new instance of that class.

702.3 XML Parser service

The current specification of JAXP has the limitation that only one of each type of parser factories can be registered. This specification specifies how multiple SAXParserFactory objects and DocumentBuilderFactory objects can be made available to bundles simultaneously.

Providers of parsers should register a JAXP factory object with the OSGi service registry under the factory class name. Service properties are used to describe whether the parser:

- Is validating
- Is name-space aware
- Has additional features

With this functionality, bundles can query the OSGi service registry for parsers supporting the specific functionality that they require.

702.4 Properties

Parsers must be registered with a number of properties that qualify the service. In this specification, the following properties are specified:

- PARSER_NAMESPACEAWARE - The registered parser is aware of name-spaces. Name-spaces allow an XML document to consist of independently developed DTDs. In an XML document, they are recognized by the xmlns attribute and names prefixed with an abbreviated name-space identifier, like: <xsl:if ...>. The type is a Boolean object that must be true when the parser supports name-spaces. All other values, or the absence of the property, indicate that the parser does not implement name-spaces.
- PARSER_VALIDATING - The registered parser can read the DTD and can validate the XML accordingly. The type is a Boolean object that must true when the parser is validating. All other values, or the absence of the property, indicate that the parser does not validate.

702.5 Getting a Parser Factory

Getting a parser factory requires a bundle to get the appropriate factory from the service registry. In a simple case in which a non-validating, non-name-space aware parser would suffice, it is best to use getServiceReference(String).

```
DocumentBuilder getParser(BundleContext context)
    throws Exception {
    ServiceReference ref = context.getServiceReference(
        DocumentBuilderFactory.class.getName() );
    if ( ref == null )
        return null;
    DocumentBuilderFactory factory =
        (DocumentBuilderFactory) context.getService(ref);
    return factory.newDocumentBuilder();
}
```

In a more demanding case, the filtered version allows the bundle to select a parser that is validating and name-space aware:

```
SAXParser getParser(BundleContext context)
    throws Exception {
    ServiceReference refs[] = context.getServiceReferences(
        SAXParserFactory.class.getName(),
            "(&(parser.namespaceAware=true)"
        +   "(parser.validating=true))" );
    if ( refs == null )
        return null;
    SAXParserFactory factory =
        (SAXParserFactory) context.getService(refs[0]);
    return factory.newSAXParser();
}
```

702.6 Adapting a JAXP Parser to OSGi

If an XML Parser supports JAXP, then it can be converted to an OSGi aware bundle by adding a BundleActivator class which registers an XML Parser Service. The utility org.osgi.util.xml.XMLParserActivator class provides this function and can be added (copied, not referenced) to any XML Parser bundle, or it can be extended and customized if desired.

702.6.1 JAR Based Services

Its functionality is based on the definition of the [5] *JAR File specification, Service Provider*. This specification defines a concept for service providers. A JAR file can contain an implementation of an abstractly defined service. The class (or classes) implementing the service are designated from a file in the META-INF/services directory. The name of this file is the same as the abstract service class.

The content of the UTF-8 encoded file is a list of class names separated by new lines. White space is ignored and the number sign ('#' \u0023) is the comment character.

JAXP uses this service provider mechanism. It is therefore likely that vendors will place these service files in the META-INF/services directory.

702.6.2 XMLParserActivator

To support this mechanism, the XML Parser service provides a utility class that should be normally delivered with the OSGi framework implementation. This class is a Bundle Activator and must start when the bundle is started. This class is copied into the parser bundle, and *not* imported.

The start method of the utility BundleActivator class will look in the META-INF/services service provider directory for the files javax.xml.parsers.SAXParserFactory (SAXFACTORYNAME) or javax.xml.parsers.DocumentBuilderFactory (DOMFACTORYNAME). The full path name is specified in the constants SAXCLASSFILE and DOMCLASSFILE respectively.

If either of these files exist, the utility BundleActivator class will parse the contents according to the specification. A service provider file can contain multiple class names. Each name is read and a new instance is created. The following example shows the possible content of such a file:

```
# ACME example SAXParserFactory file
```

```
com.acme.saxparser.SAXParserFast          # Fast
com.acme.saxparser.SAXParserValidating    # Validates
```

Both the javax.xml.parsers.SAXParserFactory and the javax.xml.parsers.DocumentBuilderFactory provide methods that describe the features of the parsers they can create. The XMLParserActivator activator will use these methods to set the values of the properties, as defined in *Properties* on page 1381, that describe the instances.

702.6.3 Adapting an Existing JAXP Compatible Parser

To incorporate this bundle activator into a XML Parser Bundle, do the following:

- If SAX parsing is supported, create a /META-INF/services/javax.xml.parsers.SAXParserFactory resource file containing the class names of the SAXParserFactory classes.
- If DOM parsing is supported, create a /META-INF/services/javax.xml.parsers.DocumentBuilderFactory file containing the fully qualified class names of the DocumentBuilderFactory classes.
- Create manifest file which imports the packages org.w3c.dom, org.xml.sax, and javax.xml.parsers.
- Add a Bundle-Activator header to the manifest pointing to the XMLParserActivator, the sub-class that was created, or a fully custom one.
- If the parsers support attributes, properties, or features that should be registered as properties so they can be searched, extend the XMLParserActivator class and override setSAXProperties(javax.xml.parsers.SAXParserFactory,Hashtable) and setDOMProperties(javax.xml.parsers.DocumentBuilderFactory,Hashtable).
- Ensure that custom properties are put into the Hashtable object. JAXP does not provide a way for XMLParserActivator to query the parser to find out what properties were added.
- Bundles that extend the XMLParserActivator class must call the original methods via super to correctly initialize the XML Parser Service properties.
- Compile this class into the bundle.
- Install the new XML Parser Service bundle.
- Ensure that the org.osgi.util.xml.XMLParserActivator class is contained in the bundle.

702.7 Usage of JAXP

A single bundle should export the JAXP, SAX, and DOM APIs. The version of contained packages must be appropriately labeled. JAXP 1.1 or later is required which references SAX 2 and DOM 2. See [4] *JAXP* for the exact version dependencies.

This specification is related to related packages as defined in the JAXP 1.1 document. The following table contains the expected minimum versions.

Table 702.1 JAXP 1.1 minimum package versions

Package	Minimum Version
javax.xml.parsers	1.1
org.xml.sax	2.0
org.xml.sax.helpers	2.0
org.xsml.sax.ext	1.0
org.w3c.dom	2.0

The Xerces project from the Apache group, [6] *Xerces 2 Java Parser*, contains a number libraries that implement the necessary APIs. These libraries can be wrapped in a bundle to provide the relevant packages.

702.8 Security

A centralized XML parser is likely to see sensitive information from other bundles. Provisioning an XML parser should therefore be limited to trusted bundles. This security can be achieved by providing ServicePermission[javax.xml.parsers.DocumentBuilderFactory| javax.xml.parsers.SAXFactory,REGISTER] to only trusted bundles.

Using an XML parser is a common function, and ServicePermission[javax.xml.parsers.DOMParserFactory|javax.xml.parsers.SAXFactory, GET] should not be restricted.

The XML parser bundle will need FilePermission[<<ALL FILES>>,READ] for parsing of files because it is not known beforehand where those files will be located. This requirement further implies that the XML parser is a system bundle that must be fully trusted.

702.9 org.osgi.util.xml

XML Parser Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.xml; version="[1.0,2.0)"

702.9.1 Summary

- XMLParserActivator - A BundleActivator class that allows any JAXP compliant XML Parser to register itself as an OSGi parser service.

702.9.2 public class XMLParserActivator implements BundleActivator, ServiceFactory<Object>

A BundleActivator class that allows any JAXP compliant XML Parser to register itself as an OSGi parser service. Multiple JAXP compliant parsers can concurrently register by using this BundleActivator class. Bundles who wish to use an XML parser can then use the framework's service registry to locate available XML Parsers with the desired characteristics such as validating and namespace-aware.

The services that this bundle activator enables a bundle to provide are:

- javax.xml.parsers.SAXParserFactory(SAXFACTORYNAME)
- javax.xml.parsers.DocumentBuilderFactory(DOMFACTORYNAME)

The algorithm to find the implementations of the abstract parsers is derived from the JAR file specifications, specifically the Services API.

An XMLParserActivator assumes that it can find the class file names of the factory classes in the following files:

- /META-INF/services/javax.xml.parsers.SAXParserFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the SAXParserFactory.
- /META-INF/services/javax.xml.parsers.DocumentBuilderFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the DocumentBuilderFactory

If either of the files does not exist, XMLParserActivator assumes that the parser does not support that parser type.

XMLParserActivator attempts to instantiate both the SAXParserFactory and the DocumentBuilderFactory. It registers each factory with the framework along with service properties:

- PARSER_VALIDATING- indicates if this factory supports validating parsers. It's value is a Boolean.
- PARSER_NAMESPACEAWARE- indicates if this factory supports namespace aware parsers It's value is a Boolean.

Individual parser implementations may have additional features, properties, or attributes which could be used to select a parser with a filter. These can be added by extending this class and overriding the setSAXProperties and setDOMProperties methods.

Concurrency Thread-safe

702.9.2.1 public static final String DOMCLASSFILE = "/META-INF/services/javax.xml.parsers.DocumentBuilderFactory"

Fully qualified path name of DOM Parser Factory Class Name file

702.9.2.2 public static final String DOMFACTORYNAME = "javax.xml.parsers.DocumentBuilderFactory"

Filename containing the DOM Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.2.3 public static final String PARSER_NAMESPACEAWARE = "parser.namespaceAware"

Service property specifying if factory is configured to support namespace aware parsers. The value is of type Boolean.

702.9.2.4 public static final String PARSER_VALIDATING = "parser.validating"

Service property specifying if factory is configured to support validating parsers. The value is of type Boolean.

702.9.2.5 public static final String SAXCLASSFILE = "/META-INF/services/javax.xml.parsers.SAXParserFactory"

Fully qualified path name of SAX Parser Factory Class Name file

702.9.2.6 public static final String SAXFACTORYNAME = "javax.xml.parsers.SAXParserFactory"

Filename containing the SAX Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.2.7 public XMLParserActivator()

702.9.2.8 public Object getService(Bundle bundle, ServiceRegistration<Object> registration)

bundle The bundle using the service.

registration The ServiceRegistration object for the service.

□ Creates a new XML Parser Factory object.

A unique XML Parser Factory object is returned for each call to this method.

The returned XML Parser Factory object will be configured for validating and namespace aware support as specified in the service properties of the specified ServiceRegistration object. This method can be overridden to configure additional features in the returned XML Parser Factory object.

Returns A new, configured XML Parser Factory object or null if a configuration error was encountered

702.9.2.9 public void setDOMProperties(DocumentBuilderFactory factory, Hashtable<String, Object> props)

factory - the DocumentBuilderFactory object

props - Hashtable of service properties.

Set the customizable DOM Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified props object.

This method can be overridden to add additional DOM2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.2.10 public void setSAXProperties(SAXParserFactory factory, Hashtable<String, Object> properties)

factory - the SAXParserFactory object

properties - the properties object for the service

Set the customizable SAX Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified properties object.

This method can be overridden to add additional SAX2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.2.11 public void start(BundleContext context) throws Exception

context The execution context of the bundle being started.

□ Called when this bundle is started so the Framework can perform the bundle-specific activities necessary to start this bundle. This method can be used to register services or to allocate any resources that this bundle needs.

This method must complete and return to its caller in a timely manner.

This method attempts to register a SAX and DOM parser with the Framework's service registry.

Throws `Exception`– If this method throws an exception, this bundle is marked as stopped and the Framework will remove this bundle's listeners, unregister all services registered by this bundle, and release all services used by this bundle.

702.9.2.12 public void stop(BundleContext context) throws Exception

context The execution context of the bundle being stopped.

□ This method has nothing to do as all active service registrations will automatically get unregistered when the bundle stops.

Throws `Exception`– If this method throws an exception, the bundle is still marked as stopped, and the Framework will remove the bundle's listeners, unregister all services registered by the bundle, and release all services used by the bundle.

702.9.2.13 public void ungetService(Bundle bundle, ServiceRegistration<Object> registration, Object service)

bundle The bundle releasing the service.

registration The `ServiceRegistration` object for the service.

service The XML Parser Factory object returned by a previous call to the `getService` method.

□ Releases a XML Parser Factory object.

702.10 References

[1] *XML*
https://www.w3.org/XML

[2] *SAX*
http://www.saxproject.org/

[3] *DOM Java Language Binding*
https://www.w3.org/TR/REC-DOM-Level-1/java-language-binding.html

[4] *JAXP*
https://www.oracle.com/java/technologies/jaxp-introduction.html

[5] *JAR File specification, Service Provider*
https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Service_Provider

[6] *Xerces 2 Java Parser*
https://xerces.apache.org/xerces2-j/

705 Promises Specification

Version 1.3

705.1 Introduction

One of the fundamental pieces of an asynchronous programming model is the mechanism by which clients retrieve the result of the asynchronous task. Since Java 5, there has been a `java.util.concurrent.Future` interface available in the Java class libraries, which means that it is the *de facto* API in Java for handling the result of an asynchronous task. Futures have some limitations however in that they have no mechanism for registering callbacks. Java 8 introduces the class `java.util.concurrent.CompletableFuture` which addresses this but it is a complex API.

This specification defines a Promises API which is independent of all other OSGi specifications including the OSGi Framework and thus can be easily used outside of the OSGi environment.

A Promise object holds the result of a potentially asynchronous task. The receiver of a Promise object can register callbacks on the Promise to be notified when the result is available or can block on the result becoming available. Promises can be chained together in powerful ways to handle asynchronous work flows and recovery.

Promises capture the effects of latency and errors by making these explicit in the API signatures. Latency is represented by callbacks which will eventually be called. Errors are represented by the failure member. In essence, this is what sets Promises apart from things such as RPC calls where such effects are not explicitly captured but rather attempted to be transparently handled.

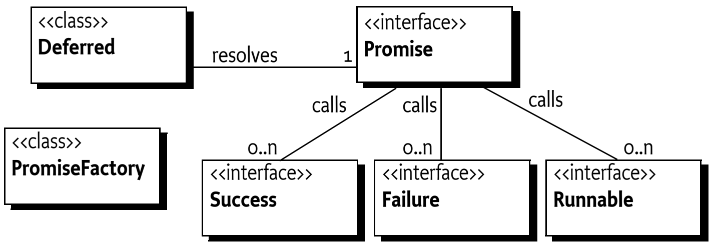
705.1.1 Essentials

- *Common concepts* - The API is inspired by the Promises work in JavaScript and uses the same basic concepts. See [2] *JavaScript Promises.*
- *Independent* - The design is independent of all other OSGi specifications and can be used outside of an OSGi environment.
- *Asynchronous* - The design supports asynchronous tasks.
- *Small* - The API and implementation are very compact.
- *Complete* - The design provides a very complete set of operations for Promise which are primitives that can be used to address most use cases.
- *Monad* - The design supports monadic programming. See [4] *Monad.*
- *Resolution* - A Promise can be resolved successfully with a value or unsuccessfully with an exception.
- *Generified* - Generics are used to promote type safety.

705.1.2 Entities

- *Promise* - A Promise object holds the eventual result of a potentially asynchronous task.
- *Callback* - The receiver of a Promise can register callbacks on the Promise to be notified when the task is completed.
- *Deferred* - A Deferred object represents the potentially asynchronous task and is used to resolve the Promise.

Figure 705.1 *Class diagram of org.osgi.util.promise*



705.2 Promise

A Promise object holds the eventual result of a potentially asynchronous task. A Promise is either unresolved or resolved. An *unresolved* Promise does not have the result of the associated task available while a *resolved* Promise has the result of the associated task available. The isDone() method must return true if the Promise is resolved and false if the Promise is unresolved. A Promise must only be resolved once.

A resolved Promise can be either resolved with a value, which means the associated task *completed successfully* and supplied a result, or resolved with a failure, which means the associated task *completed unsuccessfully* and supplied an exception. The getFailure() method can be called to determine if the resolved Promise completed successfully with a value or unsuccessfully with a failure. If the getFailure() method returns a Throwable, the Promise resolved unsuccessfully with a failure. If the getFailure() method returns null, the Promise resolved successfully with a value that can be obtained from getValue().

If the Promise is unresolved, then calling getFailure() or getValue() must block until the Promise is resolved. In general, these two methods should not be used outside of a callback. Use callbacks to be notified when the Promise is resolved. See *Callbacks* on page 1391.

705.3 Deferred

Promise is an interface which can allow for many Promise implementations. This API contains the Deferred class which provides access to the standard Promise implementation. A Deferred object can be created by calling the deferred() method on a PromiseFactory object.

A PromiseFactory object is created with a specified callback executor and a specified scheduled executor to use for created Promise objects and the Promise objects associated with created Deferred objects. If the callback executor or the scheduled executor is not specified or is specified as null, then implementation default executors will be used. The Deferred() constructor will create a Deferred whose associated Promise uses the default Promise Factory which uses the implementation default executors. All Promise objects created by a Promise must use the same Promise Factory as the creating Promise. Callbacks must be called using the callback executor of the Promise Factory associated with the Promise if the Promise Factory was created with the CALLBACKS_EXECUTOR_THREAD option. Otherwise, when the Promise is already resolved, the current thread may be used to call the callback to avoid a thread context switch. If the callback executor rejects the execution, the current thread will be used to call the callback to avoid loss of the callback operation. The scheduled executor must be used by the timeout(long) and delay(long) operations. The inlineExecutor() method can be used to obtain an executor which runs callbacks immediately on the thread calling the Executor.execute method. This behavior is similar to how callbacks were executed in the default Promise implementation of Promise 1.0 specification.

The Promise associated with a Deferred object can be obtained using getPromise(). This Promise can then be supplied to other parties who can use it to be notified of and obtain the eventual result.

```
public Promise<String> getTimeConsumingAnswer() {
  Deferred<String> deferred = factory.deferred();
  asynchronously(() -> doTask(deferred));
  return deferred.getPromise();
}
```

A Deferred object can later be used to resolve the associated Promise successfully by calling resolve(T) or unsuccessfully by calling fail(Throwable).

```
private void doTask(Deferred<String> deferred) {
  try {
    String answer = computeTimeConsumingAnswer();
    deferred.resolve(answer); // successfully resolve with value
  } catch (Exception e) {
    deferred.fail(e); // unsuccessfully resolve with exception
  }
}
```

A Deferred object can also be used to resolve the associated Promise with the eventual result of another Promise by calling resolveWith(Promise) or the result of a CompletionStage by calling resolveWith(CompletionStage).

```
private void doTask(Deferred<String> deferred) {
  try {
    Promise<String> promise = getPromiseWithTheAnswer();
    deferred.resolveWith(promise); // resolve with another Promise
  } catch (Exception e) {
    deferred.fail(e); // unsuccessfully resolve with exception
  }
}
```

If resolve(T) or fail(Throwable) is called when the Promise associated with the Deferred is already resolved, then an Illegal State Exception must be thrown.

Care must be taken in sharing a Deferred object with other parties since the other parties can resolve the associated Promise. A Deferred object should be made available only to the party that will responsible for resolving the associated Promise.

705.4 Callbacks

To be notified when a Promise has been resolved, callbacks are used. The Promise API provides two forms of callbacks: the basic Runnable and Consumer callbacks and the more specialized Success and Failure callbacks.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the callback is called.

Resolving a Promise *happens-before* any registered callback is called. That is, for the resolved Promise, in a registered callback isDone() must return true and getValue() and getFailure() must not block.

Callbacks may be registered at any time including before and after a Promise has been resolved. If a callback is registered before the Promise is resolved, it will be called later when the Promise is resolved. If a callback is registered on an already resolved Promise, it will be called right away.

705.4.1 Runnable

The onResolve(Runnable) method is used to register a Runnable with the Promise which must be called when the Promise is resolved either successfully with a value or unsuccessfully with a failure. The resolved Promise is not passed to the Runnable, so if the Runnable implementation needs access to the resolved Promise, it must take care to ensure it has access.

```
final Promise<String> answer = getTimeConsumingAnswer();
answer.onResolve(() -> doSomethingWithAnswer(answer));
```

The onResolve(Runnable) method returns the Promise object upon which it is called.

705.4.2 Consumer

The thenAccept(Consumer) method is used to register a Consumer with the Promise which must be called when the Promise is resolved successfully with a value. The value of the resolved Promise is passed to the Consumer.

```
final Promise<String> answer = getTimeConsumingAnswer().thenAccept(s ->
    doSomethingWithAnswer(s)
);
```

The thenAccept(Consumer) method returns a new Promise which will be resolved with either the exception thrown from the Consumer, if one is thrown, or with the Promise.

The onSuccess(Consumer) method is used to register a Consumer with the Promise which must be called when the Promise is resolved successfully with a value. The value of the resolved Promise is passed to the Consumer. The onSuccess(Consumer) method returns the Promise object upon which it is called.

The onFailure(Consumer) method is used to register a Consumer with the Promise which must be called when the Promise is resolved unsuccessfully with a failure. The failure of the resolved Promise is passed to the Consumer. The onFailure(Consumer) method returns the Promise object upon which it is called.

The onFailure(Consumer,Class) method is used to register a Consumer with the Promise which must be called when the Promise is resolved unsuccessfully with a failure of a specified type. The failure of the resolved Promise is passed to the Consumer. If the Promise is unsuccessfully resolved with a failure which is not of the specified type, the Consumer argument is not called. The onFailure(Consumer,Class) method returns the Promise object upon which it is called.

705.4.3 Success and Failure

The then(Success) and then(Success,Failure) methods can be used to register the more specialized Success and Failure callbacks. The Success callback is only called if the Promise is successfully resolved with a value. The Failure callback is only called if the Promise is unsuccessfully resolved with a failure.

```
Promise<String> answer = getTimeConsumingAnswer();
answer.then(p -> processResult(p.getValue()), p -> handleFailure(p.getFailure()));
```

The then methods return a new Promise which can be used to chain Promises together.

705.5 Chaining Promises

The then(Success), then(Success,Failure), and thenAccept(Consumer) methods also provide a means to chain Promises together. These methods return a new Promise which is chained to the original Promise upon which the method was called. The returned Promise must be resolved when the original Promise is resolved after the specified Success, Failure, or Consumer callback is executed. The result of the executed callback must be used to resolve the returned Promise. A sequence of calls to the then methods can be used to create a chain of promises which are resolved in sequence.

For the then(Success) or then(Success,Failure) methods, if the original Promise is successfully resolved, the Success callback is executed and the Promise returned by the Success callback, if any, or thrown exception is used to resolve the Promise returned from the method. If the original Promise is resolved with a failure, the Failure callback is executed and the Promise returned from the method is resolved with a failure.

For the thenAccept(Consumer) method, if the original Promise is successfully resolved, the Consumer callback is executed and the value of the original Promise or thrown exception is used to resolve the Promise returned from the method. If the original Promise is resolved with a failure, the Consumer callback is not executed and the Promise returned from the method is resolved with the failure of the original Promise.

In the following example, a Promise which will supply the name of the file to download is chained to a Promise which will return a mirror URL to use to download the file which is then further chained to a Promise which will return an Input Stream from which to read the download file.

```
Promise<String> name = getDownloadName();
Promise<URL> mirror = name.then(p -> getMirror(p.getValue()));
Promise<InputStream> in = mirror.then(p -> getStream(p.getValue()));
```

Since we probably do not need the intermediate Promises, we can collapse the chain into a single statement.

```
Promise<InputStream> in = getDownloadName().then(p -> getMirror(p.getValue()))
                                           .then(p -> getStream(p.getValue()));
```

The chain of Promises will also propagate any exceptions that occur to resolve the last Promise in the chain which means we do not need to do any exception handling in the intermediate tasks. Promises can also be chained by using the monadic programming methods in *Monad* on page 1393.

705.6 Monad

The Promise API supports monadic programming. See [4] *Monad*. The Promise interface defines a number of interesting methods including map, flatMap and filter.

- filter(Predicate) - Filter the value of the Promise.

  If the Promise is successfully resolved, the predicate argument is called with the value of the Promise. If the predicate accepts the value, then the value is used to successfully resolve the Promise returned by the filter method. If the predicate does not accept the value, the Promise returned by the filter method is unsuccessfully resolved with a No Such Element Exception. If the predicate throws an exception, the Promise returned by the filter method is unsuccessfully resolved with that exception.

  If the Promise is unsuccessfully resolved, the predicate argument is not called and the Promise returned by the filter method is unsuccessfully resolved with the failure of the Promise.
- map(Function) - Map the value of the Promise.

  If the Promise is successfully resolved, the function argument is called with the value of the Promise. The value returned by the function is used to successfully resolve the Promise returned by the map method. If the function throws an exception, the Promise returned by the map method is unsuccessfully resolved with that exception.

If the Promise is unsuccessfully resolved, the function argument is not called and the Promise returned by the map method is unsuccessfully resolved with the failure of the Promise.

- flatMap(Function) - FlatMap the value of the Promise.

  If the Promise is successfully resolved, the function argument is called with the value of the Promise. The Promise returned by the function is used to resolve the Promise returned by the flatMap method. If the function throws an exception, the Promise returned by the flatMap method is unsuccessfully resolved with that exception.

  If the Promise is unsuccessfully resolved, the function argument is not called and the Promise returned by the flatMap method is unsuccessfully resolved with the failure of the Promise.
- recover(Function) - Recover from the unsuccessful resolution of the Promise with a recovery value.

  If the Promise is successfully resolved, the function argument is not called and the Promise returned by the recover method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the function argument is called with the Promise to supply a recovery value. If the recovery value is not null, the Promise returned by the recover method is successfully resolved with the recovery value. If the recovery value is null, the Promise returned by the recover method is unsuccessfully resolved with the failure of the Promise. If the function throws an exception, the Promise returned by the recover method is unsuccessfully resolved with that exception.
- recover(Function,Class) - Recover from the unsuccessful resolution of the Promise with a recovery value when the failure is of a specified type.

  The recover(Function,Class) method provides the same behavior as recover(Function) but only when the failure is of the specified type. If the Promise is unsuccessfully resolved with a failure which is not of the specified type, the function argument is not called and the Promise returned by the recover method is resolved with the failure of the Promise.
- recoverWith(Function) - Recover from the unsuccessful resolution of the Promise with a recovery Promise.

  If the Promise is successfully resolved, the function argument is not called and the Promise returned by the recover method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the function argument is called with the Promise to supply a recovery Promise. If the recovery Promise is not null, the Promise returned by the recover method is resolved with the recovery Promise. If the recovery Promise is null, the Promise returned by the recover method is unsuccessfully resolved with the failure of the Promise. If the function throws an exception, the Promise returned by the recover method is unsuccessfully resolved with that exception.
- recoverWith(Function,Class) - Recover from the unsuccessful resolution of the Promise with a recovery Promise when the failure is of a specified type.

  The recoverWith(Function,Class) method provides the same behavior as recoverWith(Function) but only when the failure is of the specified type. If the Promise is unsuccessfully resolved with a failure which is not of the specified type, the function argument is not called and the Promise returned by the recover method is resolved with the failure of the Promise.
- fallbackTo(Promise) - Fall back to the value of the Promise argument if the Promise unsuccessfully resolves.

  If the Promise is successfully resolved, the Promise argument is not used and the Promise returned by the fallbackTo method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the Promise argument is used to provide a fallback value when it becomes resolved. If the Promise argument is successfully resolved, the Promise returned by the fallbackTo method is resolved with the value of the Promise argument. If the

Promise argument is unsuccessfully resolved, the Promise returned by the fallbackTo method is unsuccessfully resolved with the failure of the Promise.

- fallbackTo(Promise,Class) - Fall back to the value of the Promise argument if the Promise unsuccessfully resolves with a failure of a specified type.

  The fallbackTo(Promise,Class) method provides the same behavior as fallbackTo(Promise) but only when the failure is of the specified type. If the Promise is unsuccessfully resolved with a failure which is not of the specified type, the Promise argument is not used and the Promise returned by the fallbackTo method is resolved with the failure of the Promise.

These functions can be used to build pipelines of chained Promises that are processed in sequence. For example, in the following chain, the value of the original promise, once resolved, is filtered for acceptable values. If the filter says the value is not acceptable, the recover method will be used to replace it with a default value.

```
return promise.filter(v -> isValueOk(v)).recover(p -> getDefaultValue())
```

With these chains, one can write powerful programs without the need to resort to complex if/else and try/catch logic.

705.7 Timing

The Promise API provides methods to affect the timing of resolving Promises.

- timeout(long) - Time out the resolution of the Promise.

  If the Promise is successfully resolved before the timeout, the returned Promise is resolved with the value of the Promise. If the Promise is resolved with a failure before the timeout, the returned Promise is resolved with the failure of the Promise. If the timeout is reached before the Promise is resolved, the returned Promise is failed with a TimeoutException.
- delay(long) - Delay after the resolution of the Promise.

  Once the Promise is resolved, resolve the returned Promise with the Promise after the specified delay.

705.8 Functional Interfaces

In Java 8, the concept of Functional Interfaces is introduced. See [5] *Function Interfaces*. Functional interfaces are types with a single abstract method. Instances of functional interfaces can be created with lambda expressions, method references, or constructor references. Many methods on Promise take functional interface arguments and so are suitable for use with lambda expressions and method references in Java 8.

Four of these functional interfaces are Function, Predicate, Supplier, and Consumer. These are equivalent to functional interfaces which are part of the java.util.function package introduced in Java 8 with additional static methods to support interoperation. OSGi defines these interfaces to allow throwing checked exceptions which can be propagated in a chain of Promises.

705.9 Utility Methods

The API also provides several useful utility methods when working with Promises.

Often, you may need to create an already resolved Promise to return or chain with another Promise. The resolved(T) method can be used to create a new Promise already successfully resolved with the specified value. The failed(Throwable) method can be used to create a new Promise already unsuccessfully resolved with the specified exception. These methods also exists as static methods on the Promises class returning Promises which use the implementation default executors.

```
return getTimeConsumingAnswer().fallbackTo(factory.resolved("Fallback Value"));
```

The resolvedWith(Promise) method can be used to return a new Promise that will be resolved with the specified Promise.

The submit(Callable) method can be used to return a new Promise that will hold the result of the specified task. The task will be executed on the callback executor.

The all(Collection) method returns a Promise that is a latch on the specified Promises. The returned Promise must resolve only when all of the specified Promises have resolved. The toPromise() method returns a Collector which can be used on a Stream of Promises to collect the results of the Promises into a latch using the all(Collection) method as the finisher. The all method also exists as a static method on the Promises class returning a Promise which uses the implementation default executors.

Interoperation with CompletionStage is supported. The resolvedWith(CompletionStage) method returns a Promise that is resolved by the specified CompletionStage. The toCompletionStage() method returns a CompletionStage that is resolved by the receiving Promise.

705.10 Security

The Promise API does not define any OSGi services nor does the API perform any privileged actions. Therefore, it has no security considerations.

705.11 org.osgi.util.promise

Promise Package Version 1.3.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.util.promise; version="[1.3,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.util.promise; version="[1.3,1.4)"`

705.11.1 Summary

- `Deferred` - A Deferred Promise resolution.
- `FailedPromisesException` - Promise failure exception for a collection of failed Promises.
- `Failure` - Failure callback for a Promise.
- `Promise` - A Promise of a value.
- `PromiseFactory` - Promise factory to create Deferred and Promise objects.
- `PromiseFactory.Option` - Defines the options for a Promise factory.
- `Promises` - Static helper methods for Promises.
- `Success` - Success callback for a Promise.
- `TimeoutException` - Timeout exception for a Promise.

705.11.2 public class Deferred<T>

<T> The value type associated with the created Promise.

A Deferred Promise resolution.

Instances of this class can be used to create a Promise that can be resolved in the future. The associated Promise can be successfully resolved with resolve(Object) or resolved with a failure with fail(Throwable). It can also be resolved with the resolution of another promise using resolveWith(Promise).

The associated Promise can be provided to any one, but the Deferred object should be made available only to the party that will responsible for resolving the Promise.

Concurrency Immutable

Provider Type Consumers of this API must not implement this type

705.11.2.1 public Deferred()

□ Create a new Deferred.

The associated promise will use the default callback executor and default scheduled executor.

See Also PromiseFactory.deferred()

705.11.2.2 public void fail(Throwable failure)

failure The failure of the resolved Promise. Must not be `null`.

□ Fail the Promise associated with this Deferred.

After the associated Promise is resolved with the specified failure, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return `true` and Promise.getValue() and Promise.getFailure() must not block.

Throws `IllegalStateException`– If the associated Promise was already resolved.

705.11.2.3 public Promise<T> getPromise()

□ Returns the Promise associated with this Deferred.

All Promise objects created by the associated Promise will use the executors of the associated Promise.

Returns The Promise associated with this Deferred.

705.11.2.4 public void resolve(T value)

value The value of the resolved Promise.

□ Successfully resolve the Promise associated with this Deferred.

After the associated Promise is resolved with the specified value, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return `true` and Promise.getValue() and Promise.getFailure() must not block.

Throws `IllegalStateException`– If the associated Promise was already resolved.

705.11.2.5 public Promise<Void> resolveWith(Promise<? extends T> with)

with A Promise whose value or failure must be used to resolve the associated Promise. Must not be `null`.

□ Resolve the Promise associated with this Deferred with the specified Promise.

If the specified Promise is successfully resolved, the associated Promise is resolved with the value of the specified Promise. If the specified Promise is resolved with a failure, the associated Promise is resolved with the failure of the specified Promise.

After the associated Promise is resolved with the specified Promise, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Returns A Promise that is resolved only when the associated Promise is resolved by the specified Promise. The returned Promise must be successfully resolved with the value null, if the associated Promise was resolved by the specified Promise. The returned Promise must be resolved with a failure of IllegalStateException, if the associated Promise was already resolved when the specified Promise was resolved.

705.11.2.6 public Promise<Void> resolveWith(CompletionStage<? extends T> with)

with A CompletionStage whose result must be used to resolve the associated Promise. Must not be null.

□ Resolve the Promise associated with this Deferred with the specified CompletionStage.

If the specified CompletionStage is completed normally, the associated Promise is resolved with the value of the specified CompletionStage. If the specified CompletionStage is completed exceptionally, the associated Promise is resolved with the failure of the specified CompletionStage.

After the associated Promise is resolved with the specified CompletionStage, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Returns A Promise that is resolved only when the associated Promise is resolved by the specified CompletionStage. The returned Promise must be successfully resolved with the value null, if the associated Promise was resolved by the specified CompletionStage. The returned Promise must be resolved with a failure of IllegalStateException, if the associated Promise was already resolved when the specified CompletionStage was completed.

Since 1.2

705.11.2.7 public String toString()

□ Returns a string representation of the associated Promise.

Returns A string representation of the associated Promise.

Since 1.1

705.11.3 public class FailedPromisesException extends RuntimeException

Promise failure exception for a collection of failed Promises.

705.11.3.1 public FailedPromisesException(Collection<Promise<?>> failed, Throwable cause)

failed A collection of Promises that have been resolved with a failure. Must not be null, must not be empty and all of the elements in the collection must not be null.

cause The cause of this exception. This is typically the failure of the first Promise in the specified collection.

□ Create a new FailedPromisesException with the specified Promises.

705.11.3.2 public Collection<Promise<?>> getFailedPromises()

□ Returns the collection of Promises that have been resolved with a failure.

Returns The collection of Promises that have been resolved with a failure. The returned collection is unmodifiable.

705.11.4 public interface Failure

Failure callback for a Promise.

A Failure callback is registered with a Promise using the Promise.then(Success, Failure) method and is called if the Promise is resolved with a failure.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.11.4.1 public void fail(Promise<?> resolved) throws Exception

resolved The failed resolved Promise.

□ Failure callback for a Promise.

This method is called if the Promise with which it is registered resolves with a failure.

In the remainder of this description we will refer to the Promise returned by Promise.then(Success, Failure) when this Failure callback was registered as the chained Promise.

If this methods completes normally, the chained Promise must be failed with the same exception which failed the resolved Promise. If this method throws an exception, the chained Promise must be failed with the thrown exception.

Throws `Exception` – The chained Promise must be failed with the thrown exception.

705.11.5 public interface Promise<T>

<T> The value type associated with this Promise.

A Promise of a value.

A Promise represents a future value. It handles the interactions for asynchronous processing. A Deferred object can be used to create a Promise and later resolve the Promise. A Promise is used by the caller of an asynchronous function to get the result or handle the error. The caller can either get a callback when the Promise is resolved with a value or an error, or the Promise can be used in chaining. In chaining, callbacks are provided that receive the resolved Promise, and a new Promise is generated that resolves based upon the result of a callback.

Both callbacks and chaining can be repeated any number of times, even after the Promise has been resolved.

Example callback usage:

```
Promise<String> foo = foo();
foo.onResolve(() -> System.out.println("resolved"));
```

Example chaining usage;

```
Success<String,String> doubler = p -> Promises
  .resolved(p.getValue() + p.getValue());
Promise<String> foo = foo().then(doubler).then(doubler);
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

705.11.5.1 public Promise<T> delay(long milliseconds)

milliseconds The time to delay in milliseconds. Zero and negative time is treated as no delay.

□ Delay after the resolution of this Promise.

Once this Promise is resolved, resolve the returned Promise with this Promise after the specified delay.

Returns A Promise that is resolved with this Promise after this Promise is resolved and the specified delay has elapsed.

Since 1.1

705.11.5.2 public Promise<T> fallbackTo(Promise<? extends T> fallback)

fallback The Promise whose value must be used to resolve the returned Promise if this Promise resolves with a failure. Must not be null.

□ Fall back to the value of the specified Promise if this Promise fails.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the successful result of the specified Promise is used to resolve the returned Promise. If the specified Promise is resolved with a failure, the returned Promise must be failed with the failure of this Promise rather than the failure of the specified Promise.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise or falls back to the value of the specified Promise.

705.11.5.3 public Promise<T> fallbackTo(Promise<? extends T> fallback, Class<?> failureType)

fallback The Promise whose value must be used to resolve the returned Promise if this Promise resolves with a failure and the failure is an instance of the specified failure type. Must not be null.

failureType The type of failure for which this recovery will be used. If the failure is not an instance of the failure type, the specified recovery will not be called and the returned Promise must be failed with the failure of this Promise. Must not be null.

□ Fall back to the value of the specified Promise if this Promise fails and the failure is an instance of a failure type.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure and the failure is an instance of the specified failure type, the successful result of the specified Promise is used to resolve the returned Promise. If the specified Promise is resolved with a failure, the returned Promise must be failed with the failure of this Promise rather than the failure of the specified Promise.

If this Promise is resolved with a failure and the failure is not an instance of the specified failure type, the returned Promise must be failed with the failure of this Promise.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise or falls back to the value of the specified Promise if the failure is an instance of the specified failure type.

Since 1.3

705.11.5.4 public Promise<T> filter(Predicate<? super T> predicate)

predicate The Predicate to evaluate the value of this Promise. Must not be null.

□ Filter the value of this Promise.

If this Promise is successfully resolved, the returned Promise must either be resolved with the value of this Promise, if the specified Predicate accepts that value, or failed with a NoSuchElementException, if the specified Predicate does not accept that value. If the specified Predicate throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that filters the value of this Promise.

705.11.5.5 public Promise<R> flatMap(Function<? super T, Promise<? extends R>> mapper)

Type Parameters <R>

<R> The value type associated with the returned Promise.

mapper The Function that must flatMap the value of this Promise to a Promise that must be used to resolve the returned Promise. Must not be null.

□ FlatMap the value of this Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the Promise from the specified Function as applied to the value of this Promise. If the specified Function throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise as mapped by the specified Function.

705.11.5.6 public Throwable getFailure() throws InterruptedException

□ Returns the failure of this Promise.

If this Promise is not resolved, this method must block and wait for this Promise to be resolved before completing.

If this Promise was resolved with a failure, this method returns with the failure of this Promise. If this Promise was successfully resolved, this method must return null.

Returns The failure of this resolved Promise or null if this Promise was successfully resolved.

Throws InterruptedException– If the current thread was interrupted while waiting.

705.11.5.7 public T getValue() throws InvocationTargetException, InterruptedException

□ Returns the value of this Promise.

If this Promise is not resolved, this method must block and wait for this Promise to be resolved before completing.

If this Promise was successfully resolved, this method returns with the value of this Promise. If this Promise was resolved with a failure, this method must throw an InvocationTargetException with the failure exception as the cause.

Returns The value of this resolved Promise.

Throws InvocationTargetException– If this Promise was resolved with a failure. The cause of the InvocationTargetException is the failure exception.

InterruptedException– If the current thread was interrupted while waiting.

705.11.5.8 public boolean isDone()

□ Returns whether this Promise has been resolved.

This Promise may be successfully resolved or resolved with a failure.

Returns true if this Promise was resolved either successfully or with a failure; false if this Promise is unresolved.

705.11.5.9 public Promise<R> map(Function<? super T, ? extends R> mapper)

Type Parameters <R>

<R> The value type associated with the returned Promise.

mapper The Function that must map the value of this Promise to the value that must be used to resolve the returned Promise. Must not be null.

□ Map the value of this Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of specified Function as applied to the value of this Promise. If the specified Function throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise as mapped by the specified Function.

705.11.5.10 public Promise<T> onFailure(Consumer<? super Throwable> failure)

failure The Consumer callback that receives the failure of this Promise. Must not be null.

□ Register a callback to be called with the failure for this Promise when this Promise is resolved with a failure. The callback will not be called if this Promise is resolved successfully.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns This Promise.

Since 1.1

705.11.5.11 public Promise<T> onFailure(Consumer<? super F> failure, Class<? extends F> failureType)

Type Parameters <F>

<F> The failure type.

failure The Consumer callback that receives the failure of this Promise if the failure is an instance of the specified failure type. Must not be null.

failureType The type of failure for which this callback will be called. If the failure is not an instance of the specified failure type, the specified callback will not be called. Must not be null.

□ Register a callback to be called with the failure for this Promise when this Promise is resolved with a failure of a failure type. The callback will not be called if this Promise is resolved successfully or if the failure is not an instance of the specified failure type.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns This Promise.

Since 1.3

705.11.5.12 public Promise<T> onResolve(Runnable callback)

callback The callback to be called when this Promise is resolved. Must not be null.

□ Register a callback to be called when this Promise is resolved.

The specified callback is called when this Promise is resolved either successfully or with a failure.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns This Promise.

705.11.5.13 public Promise<T> onSuccess(Consumer<? super T> success)

success The Consumer callback that receives the value of this Promise. Must not be null.

□ Register a callback to be called with the result of this Promise when this Promise is resolved successfully. The callback will not be called if this Promise is resolved with a failure.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns This Promise.

Since 1.1

705.11.5.14 public Promise<T> recover(Function<Promise<?>, ? extends T> recovery)

recovery If this Promise resolves with a failure, the specified Function is called to produce a recovery value to be used to resolve the returned Promise. Must not be null.

□ Recover from a failure of this Promise with a recovery value.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the specified Function is applied to this Promise to produce a recovery value.

- If the recovery value is not null, the returned Promise must be resolved with the recovery value.
- If the recovery value is null, the returned Promise must be failed with the failure of this Promise.
- If the specified Function throws an exception, the returned Promise must be failed with that exception.

To recover from a failure of this Promise with a recovery value of null, the recoverWith(Function) method must be used. The specified Function for recoverWith(Function) can return Promises.resolved(null) to supply the desired null value.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise.

705.11.5.15 public Promise<T> recover(Function<Promise<?>, ? extends T> recovery, Class<?> failureType)

recovery If this Promise resolves with a failure and the failure is an instance of the specified failure type, the specified Function is called to produce a recovery value to be used to resolve the returned Promise. Must not be null.

failureType The type of failure for which the specified recovery will be used. If the failure is not an instance of the failure type, the specified recovery will not be called and the returned Promise must be failed with the failure of this Promise. Must not be null.

□ Recover from a failure of this Promise with a recovery value if the failure is an instance of a failure type.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure and the failure is not an instance of the specified failure type, the returned Promise must be failed with the failure of this Promise.

If this Promise is resolved with a failure and the failure is an instance of the specified failure type, the specified Function is applied to this Promise to produce a recovery value.

- If the recovery value is not null, the returned Promise must be resolved with the recovery value.
- If the recovery value is null, the returned Promise must be failed with the failure of this Promise.
- If the specified Function throws an exception, the returned Promise must be failed with that exception.

To recover from a failure of this Promise with a recovery value of null, the recoverWith(Function, Class) method must be used. The specified Function for recoverWith(Function, Class) can return Promises.resolved(null) to supply the desired null value.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise if the failure is an instance of the specified failure type.

Since 1.3

705.11.5.16 public Promise<T> recoverWith(Function<Promise<?>, Promise<? extends T>> recovery)

recovery If this Promise resolves with a failure, the specified Function is called to produce a recovery Promise to be used to resolve the returned Promise. Must not be null.

□ Recover from a failure of this Promise with a recovery Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the specified Function is applied to this Promise to produce a recovery Promise.

- If the recovery Promise is not null, the returned Promise must be resolved with the recovery Promise.
- If the recovery Promise is null, the returned Promise must be failed with the failure of this Promise.

- If the specified Function throws an exception, the returned Promise must be failed with that exception.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise.

705.11.5.17 public Promise<T> recoverWith(Function<Promise<?>, Promise<? extends T>> recovery, Class<?> failureType)

recovery If this Promise resolves with a failure and the failure is an instance of the specified failure type, the specified Function is called to produce a recovery Promise to be used to resolve the returned Promise. Must not be null.

failureType The type of failure for which this recovery will be used. If the failure is not an instance of the failure type, the specified recovery will not be called and the returned Promise must be failed with the failure of this Promise. Must not be null.

□ Recover from a failure of this Promise with a recovery Promise if the failure is an instance of a failure type.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure and the failure is not an instance of the specified failure type, the returned Promise must be failed with the failure of this Promise.

If this Promise is resolved with a failure and the failure is an instance of the specified failure type, the specified Function is applied to this Promise to produce a recovery Promise.

- If the recovery Promise is not null, the returned Promise must be resolved with the recovery Promise.
- If the recovery Promise is null, the returned Promise must be failed with the failure of this Promise.
- If the specified Function throws an exception, the returned Promise must be failed with that exception.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise if the failure is an instance of the specified failure type.

Since 1.3

705.11.5.18 public Promise<R> then(Success<? super T, ? extends R> success, Failure failure)

Type Parameters <R>

<R> The value type associated with the returned Promise.

success The Success callback to be called when this Promise is successfully resolved. May be null if no Success callback is required. In this case, the returned Promise must be resolved with the value null when this Promise is successfully resolved.

failure The Failure callback to be called when this Promise is resolved with a failure. May be null if no Failure callback is required.

□ Chain a new Promise to this Promise with Success and Failure callbacks.

The specified Success callback is called when this Promise is successfully resolved and the specified Failure callback is called when this Promise is resolved with a failure.

This method returns a new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success or Failure callback is executed. The result of the executed callback must be used to resolve the returned Promise. Multiple calls to this method can be used to create a chain of promises which are resolved in sequence.

If this Promise is successfully resolved, the Success callback is executed and the result Promise, if any, or thrown exception is used to resolve the returned Promise from this method. If this Promise is resolved with a failure, the Failure callback is executed and the returned Promise from this method is failed.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns A new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success or Failure callback, if any, is executed.

705.11.5.19 public Promise<R> then(Success<? super T, ? extends R> success)

Type Parameters <R>

<R> The value type associated with the returned Promise.

success The Success callback to be called when this Promise is successfully resolved. May be null if no Success callback is required. In this case, the returned Promise must be resolved with the value null when this Promise is successfully resolved.

□ Chain a new Promise to this Promise with a Success callback.

This method performs the same function as calling then(Success, Failure) with the specified Success callback and null for the Failure callback.

Returns A new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success, if any, is executed.

See Also then(Success, Failure)

705.11.5.20 public Promise<T> thenAccept(Consumer<? super T> consumer)

consumer The Consumer callback that receives the value of this Promise. Must not be null.

□ Chain a new Promise to this Promise with a Consumer callback that receives the value of this Promise when it is successfully resolved.

The specified Consumer is called when this Promise is resolved successfully.

This method returns a new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified callback is executed. If the callback throws an exception, the returned Promise is failed with that exception. Otherwise the returned Promise is resolved with the success value from this Promise.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns A new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Consumer is executed.

Since 1.1

705.11.5.21 public Promise<T> timeout(long milliseconds)

milliseconds The time to wait in milliseconds. Zero and negative time is treated as an immediate timeout.

□ Time out the resolution of this Promise.

If this Promise is successfully resolved before the timeout, the returned Promise is resolved with the value of this Promise. If this Promise is resolved with a failure before the timeout, the returned Promise is resolved with the failure of this Promise. If the timeout is reached before this Promise is resolved, the returned Promise is failed with a TimeoutException.

Returns A Promise that is resolved when either this Promise is resolved or the specified timeout is reached.

Since 1.1

705.11.5.22 public CompletionStage<T> toCompletionStage()

□ Returns a new CompletionStage that will be resolved with the result of this Promise.

Returns A new CompletionStage that will be resolved with the result of this Promise.

Since 1.2

705.11.6 public class PromiseFactory

Promise factory to create Deferred and Promise objects.

Instances of this class can be used to create Deferred and Promise objects which use the executors used to construct this object for any callback or scheduled operation execution.

Since 1.1

Concurrency Immutable

705.11.6.1 public PromiseFactory(Executor callbackExecutor)

callbackExecutor The executor to use for callbacks. null can be specified for the default callback executor.

□ Create a new PromiseFactory with the specified callback executor.

The default scheduled executor and default options will be used.

705.11.6.2 public PromiseFactory(Executor callbackExecutor, ScheduledExecutorService scheduledExecutor)

callbackExecutor The executor to use for callbacks. null can be specified for the default callback executor.

scheduledExecutor The scheduled executor for use for scheduled operations. null can be specified for the default scheduled executor.

□ Create a new PromiseFactory with the specified callback executor and specified scheduled executor.

The default options will be used.

705.11.6.3 public PromiseFactory(Executor callbackExecutor, ScheduledExecutorService scheduledExecutor, PromiseFactory.Option... options)

callbackExecutor The executor to use for callbacks. null can be specified for the default callback executor.

scheduledExecutor The scheduled executor for use for scheduled operations. null can be specified for the default scheduled executor.

options Options for PromiseFactory.

□ Create a new PromiseFactory with the specified callback executor, specified scheduled executor, and specified options.

Since 1.2

705.11.6.4 public Promise<List<T>> all(Collection<Promise<S>> promises)

Type Parameters `<T, S extends T>`

<T> The value type of the List value associated with the returned Promise.

<S> The value type of the specified Promises.

promises The Promises which must be resolved before the returned Promise must be resolved. Must not be `null` and all of the elements in the collection must not be `null`.

□ Returns a new Promise that is a latch on the resolution of the specified Promises.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

The returned Promise acts as a gate and must be resolved after all of the specified Promises are resolved.

Returns A Promise that must be successfully resolved with a List of the values in the order of the specified Promises if all the specified Promises are successfully resolved. The List in the returned Promise is the property of the caller and is modifiable. The returned Promise must be resolved with a failure of FailedPromisesException if any of the specified Promises are resolved with a failure. The failure FailedPromisesException must contain all of the specified Promises which resolved with a failure.

705.11.6.5 public Deferred<T> deferred()

Type Parameters `<T>`

<T> The value type associated with the returned Deferred.

□ Create a new Deferred with the callback executor and scheduled executor of this PromiseFactory object.

Use this method instead of Deferred.Deferred() to create a new Deferred whose associated Promise uses executors other than the default executors.

Returns A new Deferred with the callback and scheduled executors of this PromiseFactory object

705.11.6.6 public Executor executor()

□ Returns the executor to use for callbacks.

Returns The executor to use for callbacks. This will be the default callback executor if `null` was specified for the callback executor when this PromiseFactory was created.

705.11.6.7 public Promise<T> failed(Throwable failure)

Type Parameters `<T>`

<T> The value type associated with the returned Promise.

failure The failure of the resolved Promise. Must not be `null`.

□ Returns a new Promise that has been resolved with the specified failure.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

Use this method instead of Promises.failed(Throwable) to create a Promise which uses executors other than the default executors.

Returns A new Promise that has been resolved with the specified failure.

705.11.6.8 public static Executor inlineExecutor()

□ Returns an Executor implementation that executes tasks immediately on the thread calling the `Executor.execute` method.

Returns An Executor implementation that executes tasks immediately on the thread calling the Executor.execute method.

705.11.6.9 public Promise<T> resolved(T value)

Type Parameters <T>

<T> The value type associated with the returned Promise.

value The value of the resolved Promise.

□ Returns a new Promise that has been resolved with the specified value.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

Use this method instead of Promises.resolved(Object) to create a Promise which uses executors other than the default executors.

Returns A new Promise that has been resolved with the specified value.

705.11.6.10 public Promise<T> resolvedWith(CompletionStage<? extends T> with)

Type Parameters <T>

<T> The value type associated with the returned Promise.

with A CompletionStage whose result will be used to resolve the returned Promise. Must not be null.

□ Returns a new Promise that will be resolved with the result of the specified CompletionStage.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

If the specified CompletionStage is completed normally, the returned Promise is resolved with the value of the specified CompletionStage. If the specified CompletionStage is completed exceptionally, the returned Promise is resolved with the exception of the specified CompletionStage.

After the returned Promise is resolved with the specified CompletionStage, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the returned Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Returns A new Promise that will be resolved with the result of the specified CompletionStage.

Since 1.2

705.11.6.11 public Promise<T> resolvedWith(Promise<? extends T> with)

Type Parameters <T>

<T> The value type associated with the returned Promise.

with A Promise whose value or failure must be used to resolve the returned Promise. Must not be null.

□ Returns a new Promise that will be resolved with the specified Promise.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

If the specified Promise is successfully resolved, the returned Promise is resolved with the value of the specified Promise. If the specified Promise is resolved with a failure, the returned Promise is resolved with the failure of the specified Promise.

After the returned Promise is resolved with the specified Promise, all registered callbacks are called and any chained Promises are resolved. This may occur asynchronously to this method.

Resolving the returned Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Returns A new Promise that is resolved with the specified Promise.

Since 1.2

705.11.6.12 public ScheduledExecutorService scheduledExecutor()

□ Returns the scheduled executor to use for scheduled operations.

Returns The scheduled executor to use for scheduled operations. This will be the default scheduled executor if null was specified for the scheduled executor when this PromiseFactory was created.

705.11.6.13 public Promise<T> submit(Callable<? extends T> task)

Type Parameters <T>

<T> The value type associated with the returned Promise.

task The task whose result will be available from the returned Promise.

□ Returns a new Promise that will hold the result of the specified task.

The returned Promise uses the callback executor and scheduled executor of this PromiseFactory object.

The specified task will be executed on the callback executor.

Returns A new Promise that will hold the result of the specified task.

705.11.6.14 public Collector<Promise<S>, ?, Promise<List<T>>> toPromise()

Type Parameters <T, S extends T>

<T> The value type of the List value result of the collected all(Collection) Promise.

<S> The value type of the input Promises.

□ Returns a Collector that accumulates the results of the input Promises into a new all(Collection) Promise.

Returns A Collector which accumulates the results of all the input Promises into a new all(Collection) Promise.

Since 1.2

705.11.7 enum PromiseFactory.Option

Defines the options for a Promise factory.

The default options are no options unless the boolean system property org.osgi.util.promise.allowCurrentThread is set to false. When this is the case, the option Option.CALLBACKS_EXECUTOR_THREAD is a default option.

Since 1.2

705.11.7.1 CALLBACKS_EXECUTOR_THREAD

Run callbacks on an executor thread. If this option is not set, callbacks added to a resolved Promise may be immediately called on the caller's thread to avoid a thread context switch.

705.11.7.2 public static PromiseFactory.Option valueOf(String name)

705.11.7.3 public static PromiseFactory.Option[] values()

705.11.8 public class Promises

Static helper methods for Promises.

These methods return Promises which use the default callback executor and default scheduled executor. See PromiseFactory for similar methods which use executors other than the default executors.

See Also PromiseFactory

Concurrency Thread-safe

705.11.8.1 public static Promise<List<T>> all(Collection<Promise<S>> promises)

Type Parameters <T, S extends T>

<T> The value type of the List value associated with the returned Promise.

<S> A subtype of the value type of the List value associated with the returned Promise.

promises The Promises which must be resolved before the returned Promise must be resolved. Must not be null and all of the elements in the collection must not be null.

□ Returns a new Promise that is a latch on the resolution of the specified Promises.

The returned Promise acts as a gate and must be resolved after all of the specified Promises are resolved.

Returns A Promise which uses the default callback executor and default scheduled executor that is resolved only when all the specified Promises are resolved. The returned Promise must be successfully resolved with a List of the values in the order of the specified Promises if all the specified Promises are successfully resolved. The List in the returned Promise is the property of the caller and is modifiable. The returned Promise must be resolved with a failure of FailedPromisesException if any of the specified Promises are resolved with a failure. The failure FailedPromisesException must contain all of the specified Promises which resolved with a failure.

See Also PromiseFactory.all(Collection)

705.11.8.2 public static Promise<List<T>> all(Promise<? extends T>... promises)

Type Parameters <T>

<T> The value type associated with the specified Promises.

promises The Promises which must be resolved before the returned Promise must be resolved. Must not be null and all of the arguments must not be null.

□ Returns a new Promise that is a latch on the resolution of the specified Promises.

The new Promise acts as a gate and must be resolved after all of the specified Promises are resolved.

Returns A Promise which uses the default callback executor and scheduled executor that is resolved only when all the specified Promises are resolved. The returned Promise must be successfully resolved with a List of the values in the order of the specified Promises if all the specified Promises are successfully resolved. The List in the returned Promise is the property of the caller and is modifiable. The returned Promise must be resolved with a failure of FailedPromisesException if any of the specified Promises are resolved with a failure. The failure FailedPromisesException must contain all of the specified Promises which resolved with a failure.

See Also PromiseFactory.all(Collection)

705.11.8.3 public static Promise<T> failed(Throwable failure)

Type Parameters <T>

<T> The value type associated with the returned Promise.

failure The failure of the resolved Promise. Must not be null.

□ Returns a new Promise that has been resolved with the specified failure.

Returns A new Promise which uses the default callback executor and default scheduled executor that has been resolved with the specified failure.

See Also PromiseFactory.failed(Throwable)

705.11.8.4 public static Promise<T> resolved(T value)

Type Parameters <T>

<T> The value type associated with the returned Promise.

value The value of the resolved Promise.

□ Returns a new Promise that has been resolved with the specified value.

Returns A new Promise which uses the default callback executor and default scheduled executor that has been resolved with the specified value.

See Also PromiseFactory.resolved(Object)

705.11.9 public interface Success<T, R>

<T> The value type of the resolved Promise passed as input to this callback.

<R> The value type of the returned Promise from this callback.

Success callback for a Promise.

A Success callback is registered with a Promise using the Promise.then(Success) method and is called if the Promise is resolved successfully.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.11.9.1 public Promise<R> call(Promise<T> resolved) throws Exception

resolved The successfully resolved Promise.

□ Success callback for a Promise.

This method is called if the Promise with which it is registered resolves successfully.

In the remainder of this description we will refer to the Promise returned by this method as the returned Promise and the Promise returned by Promise.then(Success) when this Success callback was registered as the chained Promise.

If the returned Promise is null then the chained Promise must resolve immediately with a successful value of null. If the returned Promise is not null then the chained Promise must be resolved when the returned Promise is resolved.

Returns The Promise to use to resolve the chained Promise, or null if the chained Promise is to be resolved immediately with the value null.

Throws Exception– The chained Promise must be failed with the thrown exception.

705.11.10 public class TimeoutException extends Exception

Timeout exception for a Promise.

Since 1.1

705.11.10.1 public TimeoutException()

□ Create a new TimeoutException.

705.12 org.osgi.util.function

Function Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.function; version="[1.2,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.util.function; version="[1.2,1.3)"

705.12.1 Summary

- Consumer - A function that accepts a single argument and produces no result.
- Function - A function that accepts a single argument and produces a result.
- Predicate - A predicate that accepts a single argument and produces a boolean result.
- Supplier - A function that produces a result.

705.12.2 public interface Consumer<T>

<T> The type of the function input.

A function that accepts a single argument and produces no result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Since 1.1

Concurrency Thread-safe

705.12.2.1 public void accept(T t) throws Exception

t The input to this function.

□ Applies this function to the specified argument.

Throws Exception– An exception thrown by the method.

705.12.2.2 public Consumer<T> andThen(Consumer<? super T> after)

after The Consumer to be called after this Consumer is called. Must not be null.

□ Compose the specified Consumer to be called after this Consumer.

Returns A Consumer composed of this Consumer and the specified Consumer.

705.12.2.3 public static Consumer<T> asConsumer(Consumer<T> wrapped)

Type Parameters <T>

<T> The type of the function input.

wrapped The java.util.function.Consumer to wrap. Must not be null.

□ Returns a Consumer which wraps a java.util.function.Consumer.

Returns A Consumer which wraps the specified java.util.function.Consumer.

705.12.2.4 public static Consumer<T> asJavaConsumer(Consumer<T> wrapped)

Type Parameters <T>

<T> The type of the function input.

wrapped The Consumer to wrap. Must not be null.

□ Returns a java.util.function.Consumer which wraps the specified Consumer and throws any thrown exceptions.

The returned java.util.function.Consumer will throw any exception thrown by the wrapped Consumer.

Returns A java.util.function.Consumer which wraps the specified Consumer.

705.12.2.5 public static Consumer<T> asJavaConsumerIgnoreException(Consumer<T> wrapped)

Type Parameters <T>

<T> The type of the function input.

wrapped The Consumer to wrap. Must not be null.

□ Returns a java.util.function.Consumer which wraps the specified Consumer and discards any thrown Exceptions.

The returned java.util.function.Consumer will discard any Exception thrown by the wrapped Consumer.

Returns A java.util.function.Consumer which wraps the specified Consumer.

705.12.3 public interface Function<T, R>

<T> The type of the function input.

<R> The type of the function output.

A function that accepts a single argument and produces a result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.12.3.1 public Function<T, S> andThen(Function<? super R, ? extends S> after)

Type Parameters <S>

<S> The type of the value supplied by the specified Function.

after The Function to be called on the value returned by this Function. Must not be null.

□ Compose the specified Function to be called on the value returned by this Function.

Returns A Function composed of this Function and the specified Function.

705.12.3.2 public R apply(T t) throws Exception

t The input to this function.

□ Applies this function to the specified argument.

Returns The output of this function.

Throws Exception– An exception thrown by the method.

705.12.3.3 public static Function<T, R> asFunction(Function<T, R> wrapped)

Type Parameters <T, R>

<T> The type of the function input.

<R> The type of the function output.

wrapped The java.util.function.Function to wrap. Must not be null.

□ Returns a Function which wraps the specified java.util.function.Function.

Returns A Function which wraps the specified java.util.function.Function.

705.12.3.4 public static Function<T, R> asJavaFunction(Function<T, R> wrapped)

Type Parameters <T, R>

<T> The type of the function input.

<R> The type of the function output.

wrapped The Function to wrap. Must not be null.

□ Returns a java.util.function.Function which wraps the specified Function and throws any thrown exceptions.

The returned java.util.function.Function will throw any exception thrown by the wrapped Function.

Returns A java.util.function.Function which wraps the specified Function.

705.12.3.5 public static Function<T, R> asJavaFunctionOrElse(Function<T, R> wrapped, R orElse)

Type Parameters <T, R>

<T> The type of the function input.

<R> The type of the function output.

wrapped The Function to wrap. Must not be null.

orElse The value to return if the specified Function throws an Exception.

□ Returns a java.util.function.Function which wraps the specified Function and the specified value.

If the the specified Function throws an Exception, the the specified value is returned.

Returns A java.util.function.Function which wraps the specified Function and the specified value.

705.12.3.6 public static Function<T, R> asJavaFunctionOrElseGet(Function<T, R> wrapped, Supplier<? extends R> orElseGet)

Type Parameters <T, R>

<T> The type of the function input.

<R> The type of the function output.

wrapped The Function to wrap. Must not be null.

orElseGet The java.util.function.Supplier to call for a return value if the specified Function throws an Exception.

□ Returns a java.util.function.Function which wraps the specified Function and the specified java.util.function.Supplier.

If the the specified Function throws an Exception, the value returned by the specified java.util.function.Supplier is returned.

Returns A java.util.function.Function which wraps the specified Function and the specified java.util.function.Supplier.

705.12.3.7 public Function<S, R> compose(Function<? super S, ? extends T> before)

Type Parameters <S>

<S> The type of the value consumed the specified Function.

before The Function to be called to supply a value to be consumed by this Function. Must not be null.

□ Compose the specified Function to be called to supply a value to be consumed by this Function.

Returns A Function composed of this Function and the specified Function.

705.12.4 public interface Predicate<T>

<T> The type of the predicate input.

A predicate that accepts a single argument and produces a boolean result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.12.4.1 public Predicate<T> and(Predicate<? super T> and)

and The Predicate to be called after this Predicate is called. Must not be null.

□ Compose this Predicate logical-AND the specified Predicate.

Short-circuiting is used, so the specified Predicate is not called if this Predicate returns false.

Returns A Predicate composed of this Predicate and the specified Predicate using logical-AND.

705.12.4.2 public static Predicate<T> asJavaPredicate(Predicate<T> wrapped)

Type Parameters <T>

<T> The type of the predicate input.

wrapped The Predicate to wrap. Must not be null.

□ Returns a java.util.function.Predicate which wraps the specified Predicate and throws any thrown exceptions.

The returned java.util.function.Predicate will throw any exception thrown by the wrapped Predicate.

Returns A java.util.function.Predicate which wraps the specified Predicate.

705.12.4.3 public static Predicate<T> asJavaPredicateOrElse(Predicate<T> wrapped, boolean orElse)

Type Parameters <T>

<T> The type of the predicate input.

wrapped The Predicate to wrap. Must not be null.

orElse The value to return if the specified Predicate throws an Exception.

□ Returns a java.util.function.Predicate which wraps the specified Predicate and the specified value.

If the the specified Predicate throws an Exception, the the specified value is returned.

Returns A java.util.function.Predicate which wraps the specified Predicate and the specified value.

705.12.4.4 public static Predicate<T> asJavaPredicateOrElseGet(Predicate<T> wrapped, BooleanSupplier orElseGet)

Type Parameters <T>

<T> The type of the predicate input.

wrapped The Predicate to wrap. Must not be null.

orElseGet The java.util.function.BooleanSupplier to call for a return value if the specified Predicate throws an Exception.

□ Returns a java.util.function.Predicate which wraps the specified Predicate and the specified java.util.function.BooleanSupplier.

If the the specified Predicate throws an Exception, the value returned by the specified java.util.function.BooleanSupplier is returned.

Returns A java.util.function.Predicate which wraps the specified Predicate and the specified java.util.function.BooleanSupplier.

705.12.4.5 public static Predicate<T> asPredicate(Predicate<T> wrapped)

Type Parameters <T>

<T> The type of the predicate input.

wrapped The java.util.function.Predicate to wrap. Must not be null.

□ Returns a Predicate which wraps the specified java.util.function.Predicate.

Returns A Predicate which wraps the specified java.util.function.Predicate.

705.12.4.6 public Predicate<T> negate()

□ Return a Predicate which is the negation of this Predicate.

Returns A Predicate which is the negation of this Predicate.

705.12.4.7 public Predicate<T> or(Predicate<? super T> or)

or The Predicate to be called after this Predicate is called. Must not be null.

□ Compose this Predicate logical-OR the specified Predicate.

Short-circuiting is used, so the specified Predicate is not called if this Predicate returns true.

Returns A Predicate composed of this Predicate and the specified Predicate using logical-OR.

705.12.4.8 public boolean test(T t) throws Exception

t The input to this predicate.

□ Evaluates this predicate on the specified argument.

Returns true if the specified argument is accepted by this predicate; false otherwise.

Throws Exception– An exception thrown by the method.

705.12.5 public interface Supplier<T>

<T> The type of the function output.

A function that produces a result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.12.5.1 public static Supplier<T> asJavaSupplier(Supplier<T> wrapped)

Type Parameters <T>

<T> The type of the function output.

wrapped The Supplier to wrap. Must not be null.

□ Returns a java.util.function.Supplier which wraps the specified Supplier and throws any thrown exceptions.

The returned java.util.function.Supplier will throw any exception thrown by the wrapped Supplier.

Returns A java.util.function.Supplier which wraps the specified Supplier.

705.12.5.2 public static Supplier<T> asJavaSupplierOrElse(Supplier<T> wrapped, T orElse)

Type Parameters <T>

<T> The type of the function output.

wrapped The Supplier to wrap. Must not be null.

orElse The value to return if the specified Supplier throws an Exception.

□ Returns a java.util.function.Supplier which wraps the specified Supplier and the specified value.

If the the specified Supplier throws an Exception, the the specified value is returned.

Returns A java.util.function.Supplier which wraps the specified Supplier and the specified value.

705.12.5.3 public static Supplier<T> asJavaSupplierOrElseGet(Supplier<T> wrapped, Supplier<? extends T> orElseGet)

Type Parameters <T>

<T> The type of the function output.

wrapped The Supplier to wrap. Must not be null.

orElseGet The java.util.function.Supplier to call for a return value if the specified Supplier throws an Exception.

□ Returns a java.util.function.Supplier which wraps the specified Supplier and the specified java.util.function.Supplier.

If the the specified Supplier throws an Exception, the value returned by the specified java.util.function.Supplier is returned.

Returns A java.util.function.Supplier which wraps the specified Supplier and the specified java.util.function.Supplier.

705.12.5.4 public static Supplier<T> asSupplier(Supplier<T> wrapped)

Type Parameters <T>

<T> The type of the function output.

wrapped The java.util.function.Supplier to wrap. Must not be null.

□ Returns a Supplier which wraps the specified java.util.function.Supplier.

Returns A Supplier which wraps the specified java.util.Supplier.Function.

705.12.5.5 public T get() throws Exception

□ Returns a value.

Returns The output of this function.

Throws Exception– An exception thrown by the method.

705.13 References

[1] *JavaScript Promises*
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[2] *JavaScript Promises*
https://web.dev/promises/

[3] *ECMAScript® 2022 Language Specification*
https://tc39.es/ecma262/#sec-promise-objects

[4] *Monad*
https://en.wikipedia.org/wiki/Monad_%28functional_programming%29

[5] *Function Interfaces*
https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8

705.14 Changes

- Added new overloaded methods onFailure(Consumer,Class), recover(Function,Class), recoverWith(Function,Class), and fallbackTo(Promise,Class). These overloaded methods allow the caller to specify the type of the failure for which they are in effect.

706 Push Stream Specification

Version 1.1

706.1 Introduction

In large-scale distributed systems *events* are a commonly used communication mechanism for passing data and triggering behaviors. Events are typically generated *asynchronously* rather than at the request of the processing system, and once received an event usually undergoes some level of transformation before being stored, acted upon, or forwarded to another consumer.

Pipelines and streams are a popular and effective model for consuming and processing events, with numerous APIs providing this sort of model. One of the most well-known processing pipeline APIs is the Java 8 Streams API, which provides a functional pipeline for operating on Collections. The Streams API is inherently *pull based*" as it relies on iterators and spliterators to *pull* the next entry from the stream. This is the primary difference between synchronous and asynchronous models. In an asynchronous world events are pushed into the pipeline as they are received.

This specification defines a PushStream API which can be used on devices which support the Java 8 compact1 profile. The PushStream API defined by this specification depends on OSGi Promises but is independent of all other OSGi specifications, including the OSGi Framework, and thus can be easily used outside of the OSGi environment.

A PushStream object encapsulates a pipeline of a potentially asynchronous tasks which will be performed when an event arrives. The result of the processing pipeline is represented using a Promise object which will resolve when the result has been calculated.

PushStream capture the effects of errors, finite streams and back pressure by making these explicit in the API signatures. Errors and End of Stream conditions are represented by specific events which are pushed into the stream. Back pressure is represented by a delay value returned from the event pipeline stages.

706.1.1 Essentials

- *Common concepts* - The API is inspired by the Streams API in Java 8 and uses the same basic concepts. See [1] *Java 8 Stream API.*
- *Independent* - The design is independent of all other OSGi specifications (except for OSGi Promises) and can be used outside of an OSGi environment.
- *Asynchronous* - The design is built to handle asynchronously produced events.
- *Back Pressure* - The design provides a means for event pipelines to communicate back-pressure to the Event Source.
- *Complete* - The design provides a very complete set of operations for PushStreams which are primitives that can be used to address most use cases.
- *Generified* - Generics are used to promote type safety.

706.1.2 Entities

- *Push Event Source* - A PushEventSource object represents a source of asynchronous events, and can be used to create a PushStream.

- *Push Event Consumer* - A Push Event Consumer object represents a sink for asynchronous events, and can be attached to a PushEventSource or a PushStream.
- *Push Stream* - A PushStream object represents a pipeline for processing asynchronous events.
- *Terminal Operation* - The final operation of a PushStream pipeline results in a Promise which represents the completion state of the pipeline. The operation also begins the processing of events.

706.2 Asynchronous Event Streams

The Push Stream API is built upon the principals of Asynchronous Event streams, and therefore requires three basic primitives:

- An event object
- A source of event objects
- A consumer of event objects

706.2.1 The Push Event

The `PushEvent` is an object representing an event. Every Push Event has an event type, which has one of three values:

- `DATA` - A data event encapsulates a typed object
- `ERROR` - An error event encapsulates an exception and indicates a failure in the event stream.
- `CLOSE` - A close event represents the end of the stream of events.

An event stream consists of zero or more data events followed by a *terminal event*. A terminal event is either an error or a close, and it indicates that there will be no more events in this stream. Depending on the reason for the terminal event it may be possible to re-attach to the event source and consume more events.

706.2.2 The Push Event Source

A Push Event Source object represents a source of asynchronous Push Events. The event source defines a single method `open(PushEventConsumer)` which can be used to connect to the source and begin receiving a stream of events.

The `open` method of the Push Event Source returns an `AutoCloseable` which can be used to close the event stream. If the `close` method is called on this object then the stream is terminated by sending a close event. If additional calls are made to the close method then they return without further action.

706.2.3 The Push Event Consumer

A Push Event Consumer object represents a sink for asynchronous Push Events. The event consumer defines a single method `accept(PushEvent)` which can be used to receive a stream of events.

The `accept` method of the Push Event Consumer returns a `long` representing *back pressure*. Back pressure is described in detail in *Back pressure* on page 1428. If the returned long is negative then the event stream should be closed by the event source.

706.2.4 Closing the Event Stream

There are three ways in which a stream of events can complete normally.

- The Push Event Source may close the stream at any time by sending a terminal event to the consumer. Upon receiving a terminal event the consumer should clean up any resources and not expect to receive further messages. Note that in a multi-threaded system the consumer may receive events out of order, and in this case data events may be received after a terminal event. Event processors should be careful to ignore data events that occur after terminal events, and to ensure that any downstream consumers receive any pending data events before forwarding the terminal event.

- The open method of the Push Event Source returns an AutoCloseable which can be used to close the event stream. If the close method is called on this object then the stream is terminated by sending a close event. If additional calls are made to the close method then they return without action. If the close method is called after a terminal event has been sent for any other reason then it must return without action.
- The accept method of the Push Event Consumer returns a long indicating back pressure. If the long is negative then the event source must close the stream by sending a close event.

706.3 The Push Stream

Simple event passing can be achieved by connecting a Push Event Consumer directly to a Push Event Source, however this model forces a large amount of flow-control and resource management into a single location. Furthermore it is difficult to reuse business logic across different event streams.

The PushStream provides a powerful, flexible pipeline for event processing. The Push Stream API shares many concepts with the Java 8 Streams API, in particular Push Streams are lazy, they may not consume the entire event stream, and they can be composed from functional steps.

706.3.1 Simple Pipelines

A Push Stream can be created from a Push Event Source by using a PushStreamProvider. A Push Stream represents a stage in an event processing pipeline. The overall pipeline is constructed from zero or more *intermediate operations*, and completed with a single *terminal operation*.

Each intermediate operation returns a new Push Stream object chained to the previous pipeline step. Once a Push Stream object has had an intermediate operation invoked on it then it may not have any other operations chained to it. Terminal operations are either void, or return a Promise representing the future result of the pipeline. These API patterns allow Push Streams to be built using a fluent API.

Push Stream instances are lazy, and so the Push Stream will not be connected to the Push Event Source until a terminal operation is invoked on the Push Stream. This means that a push stream object can be safely built without events being received when the pipeline is partially initialized.

706.3.1.1 Mapping, Flat Mapping and Filtering

The simplest intermediate operations on a Push Stream are *mapping* and *filtering*. These operations use stateless, non-interfering functions to alter the data received by the next stage in the pipeline.

706.3.1.1.1 Mapping

Mapping is the act of transforming an event from one type into another. This may involve taking a field from the object, or performing some simple processing on it. When mapping there is an *one to one* relationship between input and output events, that is, each input event is mapped to exactly one output event.

```
PushStream<String> streamOfStrings = getStreamOfStrings();

PushStream<Integer> streamOfLengths =
        streamOfStrings.map(String::length);
```

If the mapping function throws an Exception then an Error Event is propagated down the stream to the next pipeline step. The failure in the error event is set to the Exception thrown by the mapping function. The current pipeline step is also closed, and the close operation is propagated back upstream to the event source by closing previous pipeline stages. Any subsequently received events must not be propagated and must return negative back pressure.

706.3.1.1.2 Flat Mapping

Flat Mapping is the act of transforming an event from one type into multiple events of another type. This may involve taking fields from an object, or performing some simple processing on it. When flat mapping there is a *one to many* relationship between input and output events, that is, each input event is mapped to zero or more output events.

A flat mapping function should asynchronously consume the event data and return a Push Stream containing the flow of subsequent events.

```
PushStream<String> streamOfStrings = getStreamOfStrings();

PushStream<Character> streamOfCharacters =
        streamOfStrings.flatMap(s -> {
                SimplePushEventSource<Character> spes =
                        getSimplePushEventSource();

                spes.connectPromise()
                    .onResolve(() ->
                        executor.execute(() -> {
                                for(int i = 0; i < s.length; i++) {
                                    spes.publish(s.charAt(i));
                                }
                            });
                return pushStreamProvider.createStream(spes);
            });
```

If the flat mapping function throws an Exception then an Error Event is propagated down the stream to the next pipeline step. The failure in the error event is set to the Exception thrown by the mapping function. The current pipeline step is also closed, and the close operation is propagated back upstream to the event source by closing previous pipeline stages. Any subsequently received events must not be propagated and must return negative back pressure.

706.3.1.1.3 Filtering

Filtering is the act of removing events from the stream based on some characteristic of the event data. This may involve inspecting the fields of the data object, or performing some simple processing on it. If the filter function returns true for an event then it will be passed to the next stage of the pipeline. If the filter function returns false then it will be discarded, and not passed to the next pipeline stage.

```
PushStream<String> streamOfStrings = getStreamOfStrings();

PushStream<String> filteredStrings =
        streamOfStrings.filter(s -> s.length() == 42);
```

If the filtering function throws an Exception then an Error Event is propagated down the stream to the next pipeline step. The failure in the error event is set to the Exception thrown by the filter function. The current pipeline step is also closed, and the close operation is propagated back upstream to the event source by closing previous pipeline stages. Any subsequently received events must not be propagated and must return negative back pressure.

706.3.1.1.4 Asynchronous Mapping

Mapping operations may sometimes take time to calculate their results. PushStream operations should, in general be fast and non-blocking and so long-running mapping operations should be run on a separate thread. The asyncMap(int,int,Function) operation allows the mapping function to return a Promise representing the ongoing calculation of the mapped value. When this promise resolves then its value will be passed to the next pipeline stage.

As asynchronous mapping operations are long-running they require back pressure to be generated as the number of running operations increases. The amount of back pressure returned is equal to the number of pending promises (aside from the mapping operation that has just started) plus the number of waiting threads if the maximum number of concurrent promises has been reached. The returned back pressure when only a single promise is running is therefore always zero.

706.3.1.2 Stateless and Stateful Intermediate Operations

Intermediate operations are either *stateless* or *stateful*. Stateless operations are ones where the pipeline stage does not need to remember the previous data from the stream. Mapping, Flat Mapping and Filtering are all stateless operations. The following table lists the stateless operations on the Push Stream.

Table 706.1 Stateless Intermediate Operations on the Push Stream

Intermediate Operation	Description

adjustBackPressure(LongUnaryOperator)
adjustBackPressure(ToLongBiFunction)
	Register a transformation function to adjust the back pressure returned by the previous entry in the stream. The result of this function will be returned as back pressure.

asyncMap(int,int,Function)	Register a mapping function which will asynchronously calculate the value to be passed to the next stage of the stream. The returned back pressure is equal to one less than the number of outstanding promises, plus the number of queued threads, multiplied by the delay value.
filter(Predicate)	Register a selection function to be called with each data event in the stream. If the function returns true then the event will propagated, if false then the event will dropped from the stream.
flatMap(Function)	Register a transformation function to be called with each data event in the stream. Each incoming data element is converted into a stream of elements. The transformed data is then propagated to the next stage of the stream.
fork(int,int,Executor)	Pushes event processing onto one or more threads in the supplied Executor returning a fixed back pressure
map(Function)	Register a transformation function to be called with each data event in the stream. The transformed data is propagated to the next stage of the stream.
merge(PushStream)	Merges this stream and another stream into a single stream. The returned stream will not close until both parent streams are closed.
sequential()	Forces data events to be delivered sequentially to the next stage of the stream. Events may be delivered on multiple threads, but will not arrive concurrently at the next stage of the pipeline.
split(Predicate...)	Register a set of filter functions to select elements that should be forwarded downstream. The returned streams correspond to the supplied filter functions.

Stateful operations differ from stateless operations in that they must remember items from the stream. Sometimes stateful operations must remember large numbers of events, or even the entire stream. For example the distinct operation remembers the identity of each entry in the stream, and filters out duplicate events.

Care should be taken when using Stateful operations with large or infinite streams. For example the sorted operation must process the *entire* stream until it receives a close event. At this point the events can be sorted and delivered in order. It is usually a good idea to use the limit operation to restrict the length of the stream before performing a stateful operation which must remember many elements.

The following table lists all of the stateful operations of the PushStream.

Table 706.2 Stateful Intermediate Operations on the Push Stream

Intermediate Operation	Description
buffer()	Introduces a buffer before the next stage of the stream. The buffer can be used to provide a circuit breaker, or to allow a switch of consumer thread(s).
buildBuffer()	Introduces a configurable buffer before the next stage of the stream. The buffer can be used to provide a circuit breaker, or to allow a switch of consumer thread(s).

coalesce(Function)
coalesce(int,Function)
coalesce(IntSupplier,Function)
	Register a coalescing function which aggregates one or more data events into a single data event which will be passed to the next stage of the stream.
	
	The number of events to be accumulated is either provided as a fixed number, or as the result of a function

distinct()	A variation of filter(Predicate) which drops data from the stream that has already been seen. Specifically if a data element equals an element which has previously been seen then it will be dropped. This stateful operation must remember all data that has been seen.
limit(long)	Limits the length of the stream to the defined number of elements. Once that number of elements are received then a close event is propagated to the next stage of the stream.
limit(Duration)	Limits the time that the stream will remain open to the supplied Duration. Once that time has elapsed then a close event is propagated to the next stage of the stream.
skip(long)	Drops the supplied number of data events from the stream and then forwards any further data events.

sorted()
sorted(Comparator)
	Remembers all items in the stream until the stream ends. At this point the data in the stream will be propagated to the next stage of the stream, either in the Natural Ordering of the elements, or in the order defined by the supplied Comparator.

timeout(Duration)	Tracks the time since the last event was received. If no event is received within the supplied Duration then an error event is propagated to the next stage of the stream. The exception in the event will be an org.osgi.util.promise.TimeoutException.

window(Duration,Function)
window(Duration,Executor,Function)
window(Supplier,IntSupplier,BiFunction)
window(Supplier,IntSupplier,Executor,BiFunction)
	Collects events over the specified time-limit, passing them to the registered handler function. If no events occur during the time limit then a Collection containing no events is passed to the handler function.

706.3.1.3 Terminal Operations

Terminal operations mark the end of a processing pipeline. Invoking a terminal operation causes the PushStream to connect to its underlying event source and begin processing.

The simplest terminal operation is the count() operation. This method returns a promise that will resolve when the stream finishes. If the stream finishes with a close event then the promise will resolve with a Long representing the number of events that reached the end of the pipeline. If the stream finishes with an error then the promise will fail with that error.

Terminal operations such as forEachEvent(PushEventConsumer) are passed a handler function which will be called for each piece of data that reaches the end of the stream. If the handler function throws an Exception then the Promise returned by the terminal operation must fail with the Exception thrown by the handler function.

Some terminal operations, like count require the full stream to be processed, others are able to finish before the end of the stream. These are known as *short circuiting* operations. An example of a short-circuiting operation is findFirst(). This operation resolves the promise with the first event that is received by the end of the pipeline. Once a short-circuiting operation has completed it propagates negative back-pressure through the pipeline to close the source of events. Any subsequently received events must not affect the result and must return negative back pressure. If an asynchronous pipeline step is encountered, such as a buffer, the close operation is propagated back upstream to the event source by closing previous pipeline stages.

Table 706.3 Non Short Circuiting Terminal Operations on the Push Stream

Terminal Operation	**Description**
collect(Collector)	Uses the Java Collector API to collect the data from events into a single Collection, Map, or other type.
count()	Counts the number of events that reach the end of the stream pipeline.
forEach(Consumer)	Register a function to be called back with the data from each event in the stream
forEachEvent(PushEventConsumer)	Register a PushEventConsumer to be called back with each event in the stream. If negative back-pressure is returned then the stream will be closed.
max(Comparator)	Uses a Comparator to find the largest data element in the stream of data. The promise is resolved with the final result when the stream finishes.
min(Comparator)	Uses a Comparator to find the smallest data element in the stream of data. The promise is resolved with the final result when the stream finishes.

reduce(BinaryOperator)
reduce(T,BinaryOperator)
reduce(U,BiFunction,BinaryOperator)
	Uses a Binary Operator function to combine event data into a single object. The promise is resolved with the final result when the stream finishes.

toArray()
toArray(IntFunction)
	Collects together all of the event data in a single array which is used to resolve the returned promise.

Table 706.4 Short Circuiting Terminal Operations on the Push Stream

Terminal Operation	**Description**
allMatch(Predicate)	Resolves with false if any event reaches the end of the stream pipeline that does not match the predicate. If the stream ends without any data matching the predicate then the promise resolves with true
anyMatch(Predicate)	Resolves with true if any data event reaches the end of the stream pipeline and matches the supplied predicate. If the stream ends without any data matching the predicate then the promise resolves with false

Terminal Operation	Description
findAny()	Resolves with an Optional representing the data from the first event that reaches the end of the pipeline. If the stream ends without any data reaching the end of the pipeline then the promise resolves with an empty Optional.
findFirst()	Resolves with an Optional representing the data from the first event that reaches the end of the pipeline. If the stream ends without any data reaching the end of the pipeline then the promise resolves with an empty Optional.
noneMatch(Predicate)	Resolves with false if any data event reaches the end of the stream pipeline and matches the supplied predicate. If the stream ends without any data matching the predicate then the promise resolves with true

706.3.2 Buffering, Back pressure and Circuit Breakers

Buffering and Back Pressure are an important part of asynchronous stream processing. Back pressure and buffering are therefore an important part of the push stream API.

706.3.2.1 Back pressure

In a synchronous model the producer's thread is held by the consumer until the consumer has finished processing the data. This is not true for asynchronous systems, and so a producer can easily overwhelm a consumer with data. Back pressure is therefore used in asynchronous systems to allow consumers to control the speed at which producers provide data.

Back pressure in the asynchronous event processing model is provided by the PushEventConsumer. The value returned by the accept method of the PushEventConsumer is an indication of the requested back pressure. A return of zero indicates that event delivery may continue immediately. A positive return value indicates that the source should delay sending any further events for the requested number of milliseconds. A negative return value indicates that no further events should be sent and that the stream can be closed.

Back pressure in a Push Stream can also be applied mid-way through the processing pipeline through the use of the adjustBackPressure(LongUnaryOperator) or adjustBackPressure(ToLongBiFunction) methods. These methods can be used to increase or decrease the back pressure requested by later stages of the pipeline.

706.3.2.2 Buffering

In asynchronous systems events may be produced and consumed at different rates. If the consumer is faster than the producer then there is no issue, however if the producer is faster than the consumer then events must be held somewhere. Back pressure provides some assistance here, however some sources do not have control over when events are produced. In these cases the data must be buffered until it can be processed.

As well as providing a queue for pending work, introducing buffers allows event processing to be moved onto a different thread, and for the number of processing threads to be changed part way through the pipeline. Buffering can therefore protect an PushEventSource from having its event generation thread “stolen” by a consumer which executes a long running operation. As a result the PushEventSource can be written more simply, without a thread switch, if a buffer is used.

Buffering also provides a “fire break” for back-pressure. Back-pressure return values propagate back along a PushStream until they reach a part of the stream that is able to respond. For some PushEventSource implementations it is not possible to slow or delay event generation, however a buffer can always respond to back pressure by not releasing events from the buffer. Buffers can therefore be used to “smooth out” sources that produce bursts of events more quickly than they can be immediately processed. This simplifies the creation of PushEventConsumer instances, which can rely on their back-pressure requests being honored.

Buffering is provided by the Push Stream using default configuration values, either when creating the Push Stream from the Push Stream Provider, or using the buffer method. These defaults are described in *Building a Buffer or Push Stream* on page 1429.

The default configuration values can be overridden by using a BufferBuilder to explicitly provide the buffering parameters. If no Executor is provided then the PushStream will create its own internal Executor with the same number of threads as the defined parallelism. An internally created Executor will be shut down when the PushStream is closed.

706.3.2.3 Buffering policies

Buffering policies govern the behavior of a buffer as it becomes full.

The QueuePolicy of the buffer determines what happens when the queue becomes full. Different policies may discard incoming data, evict data from the buffer, block, or throw an exception.

The QueuePolicyOption provides basic implementations of the queue policies, but custom polices can be implemented to provide more complex behaviors.

The PushbackPolicy of the buffer determines how much back pressure is requested by the buffer. Different policies may return a constant value, slowly increase the back pressure as the buffer fills, or return an exponentially increasing value when the buffer is full.

The PushbackPolicyOption provides basic implementations of the push back policies, but custom polices can be implemented to provide more complex behaviors.

The ThresholdPushbackPolicy provides an implementation of a push back policy covering a common use case. In this use case back pressure needs to be applied only after the number of items in the buffer passes a certain threshold. Once this threshold is exceeded then the back pressure may be fixed, or may increase linearly based on the number of buffered items.

706.3.2.4 Building a Buffer or Push Stream

The PushStreamBuilder can be obtained from a Push Stream Provider and used to customize the buffer at the start of the PushStream, or it can be used to create an unbuffered PushStream. An unbuffered PushStream uses the incoming event delivery thread to process the events, and therefore users must be careful not to block the thread, or perform long-running tasks. The default configuration building a Push Stream is as follows:

- A parallelism of one
- A FAIL queue policy
- A LINEAR push back policy with a maximum push back of one second
- A Buffer with a capacity of 32 elements

A Push Stream also requires a timer and an executor. For a new Push Stream the Push Stream Provider must create a new fixed pool of worker threads with the same size as the parallelism. The Push Stream Provider may create a new ScheduledExecutorService for each new Push Stream, or reuse a common Scheduler. When adding a buffer to an existing Push Stream the existing executor and timer used by the Push Stream are reused by default. The builder of the Buffer/ Push Stream may provide their own executor and timer using the withExecutor(Executor) and withScheduler(ScheduledExecutorService) methods

706.3.2.5 Circuit Breakers

Buffering is a powerful tool in event processing pipelines, however it cannot help in the situation where the average event production rate is higher than the average processing rate. Rather than having an infinitely growing buffer a circuit breaker is used. A circuit breaker is a buffer which fails the stream when the buffer is full. This halts event processing and prevents the consuming system from being overwhelmed.

The default policy for push stream buffers is the FAIL policy, which means that push stream buffers are all circuit breakers by default.

706.3.3 Forking

Sometimes the processing that needs to be performed on an event is long-running. An important part of the asynchronous eventing model is that callbacks are short and non-blocking, which means that these callbacks should not run using the primary event thread. One solution to this is to buffer the stream, allowing a thread handoff at the buffer and limiting the impact of the long-running task. Buffering, however, has other consequences, and so it may be the case that a simple thread hand-off is preferable.

Forking allows users to specify a maximum number of concurrent downstream operations. Incoming events will block if this limit has been reached. If there are blocked threads then the returned back pressure for an event will be equal to the number of queued threads multiplied by the supplied timeout value. If there are no blocked threads then the back pressure will be zero.

706.3.4 Coalescing and Windowing

Coalescing and windowing are both processes by which multiple incoming data events are collapsed into a single outgoing event.

706.3.4.1 Coalescing

There are two main ways to coalesce a stream.

The first mechanism delegates all responsibility to the coalescing function, which returns an `Optional`. The coalescing function is called for every data event, and returns an optional which either has a value, or is empty. If the optional has a value then this value is passed to the next stage of the processing pipeline. If the optional is empty then no data event is passed to the next stage.

The second mechanism allows the stream to be configured with a (potentially variable) buffer size. The stream then stores values into this buffer. When the buffer is full then the stream passes the buffer to the handler function, which returns data to be passed to the next stage. If the stream finishes when a buffer is partially filled then the partially filled buffer will be passed to the handler function.

When coalescing events there is no opportunity for feedback from the event handler while the events are being buffered. As a result back pressure from the handler is zero except when the event triggers a call to the next stage. When the next stage is triggered the back pressure from that stage is returned.

706.3.4.2 Windowing

Windowing is similar to coalescing, the primary difference between coalescing and windowing is the way in which the next stage of processing is triggered. A coalescing stage collects events until it has the correct number and then passes them to the handler function, regardless of how long this takes. A windowing stage collects events for a given amount of time, and then passes the collected events to the handler function, regardless of how many events are collected.

To avoid the need for a potentially infinite buffer a windowing stage may also place a limit on the number of events to be buffered. If this limit is reached then the window finishes early and the buffer is passed to the client, just like a coalescing stage. In this mode of operation the handler function is also passed the length of time for which the window lasted.

As windowing requires the collected events to be delivered asynchronously there is no opportunity for back-pressure from the previous stage to be applied upstream. Windowing therefore returns zero back-pressure in all cases except when a buffer size limit has been declared and is reached. If a window size limit is reached then the windowing stage returns the remaining window time as back pressure. Applying back pressure in this way means that the event source will tend not to repeatedly over saturate the window.

706.3.5 Merging and Splitting

Merging and Splitting are actions that can be used to combine push streams, or to convert one stream into many streams.

706.3.5.1 Merging

A client may need to consume data from more than one Event Sources. In this case the PushStream may be used to merge two event streams. The returned stream will receive events from both parent streams, but will only close when *both* parent streams have delivered terminal events.

706.3.5.2 Splitting

Sometimes it is desirable to split a stream into multiple parallel pipelines. These pipelines are independent from the point at which they are split, but share the same source and upstream pipeline.

Splitting a stream is possible using the `split(Predicate<? super T >... predicates)` method. For each predicate a PushStream will be returned that receives the events accepted by the predicate.

The lifecycle of a split stream differs from that of a normal stream in two key ways:

- The stream will begin event delivery when any of the downstream handlers encounters a terminal operation
- The stream will only close when all of the downstream handlers are closed

706.3.6 Time Limited Streams

An important difference between Push Streams and Java 8 Streams is that events occur over time, there are therefore some operations that do not apply to Java 8 Streams which are relevant to Push Streams.

The `limit()` operation on a Stream can be used to limit the number of elements that are processed, however on a Push Stream that number of events may never be reached, even though the stream has not closed. Push Streams therefore also have a `limit` method which takes a `Duration`. This duration limits the time for which the stream is open, closing it after the duration has elapsed.

The `timeout` operation of a Push Stream can be used to end a stream if no events are received for the given amount of time. If an event is received then this resets the timeout counter. The timeout operation is therefore a useful mechanism for identifying pipelines which have stalled in their processing. If the timeout expires then it propagates an error event to the next stage of the pipeline. The Exception in the error event is an `org.osgi.util.promise.TimeoutException`.

706.3.7 Closing Streams

A `PushStream` represents a stage in the processing pipeline and is `AutoCloseable`. When the `close()` method is invoked it will not, in general, coincide with the processing of an event. The closing of a stream in this way must therefore do the following things:

- Send a close event downstream to close the stream
- Discard events subsequently received by this pipeline stage, and return negative backpressure for any that do arrive at this pipeline stage.
- Propagate the close operation upstream until the `AutoCloseable` returned by the `open(PushEventConsumer)` method is closed.

The result of this set of operations must be that all stages of the pipeline, including the connection to the `PushEventSource`, are eagerly closed. This may be as a result of receiving a close event, negative back pressure, or the close call being propagated back up the pipeline, but it must not wait for the next event. For example, if an event is produced every ten minutes and the stream is closed one minute after an event is created then it must not take a further nine minutes to close the connection to the Push Event Source.

706.4 The Push Stream Provider

The PushStreamProvider can be used to assist with a variety of asynchronous event handling use cases. A Push Stream Provider can create Push Stream instances from a Push Event Source, it can buffer an Push Event Consumer, or it can turn a Push Stream into a reusable Push Event Source.

706.4.1 Building Buffers

The Push Stream Provider allows several types of buffered objects to be created. By default all Push Streams are created with a buffer, but other objects can also be wrapped in a buffer. For example a Push Event Consumer can be wrapped in a buffer to isolate it from a Push Event Source. The SimplePushEventSource also has a buffer, which is used to isolate the event producing thread from event consumers.

In all cases buffers are configured using a BufferBuilder with the following defaults:

- A parallelism of one
- A FAIL QueuePolicy
- A LINEAR PushbackPolicy with a maximum pushback of one second
- A Buffer with a capacity of 32 elements

A Buffer requires a timer and an executor. If no Executor is provided when creating a buffer then the buffer will have its own internal Executor with the same number of threads as the defined parallelism. The Push Stream Provider may create a new ScheduledExecutorService for each buffer, or reuse a common Scheduler. The builder of the Buffer may provide their own executor and timer using the withExecutor(Executor) and withScheduler(ScheduledExecutorService) methods

Any internally created Executor will be shut down after the buffer has processed a terminal event.

706.4.2 Mapping between Java 8 Streams and Push Streams

There are a number of scenarios where an application developer may wish to convert between a Java 8 Stream and a PushStream. In particular, the flatMap(Function) operation of a Push Stream takes a single event and converts it into many events in a Push Stream. Common operations, such as splitting the event into child events will result in a Java Collection, or a Java 8 Stream. These need to be converted into a Push Stream before they can be returned from the flatMap operation.

To assist this model the PushStreamProvider provides two streamOf methods. These convert a Java 8 Stream into a Push Stream, changing the pull-based model of Java 8 Streams into the asynchronous model of the Push Stream.

The first streamOf(Stream) method takes a Java 8 Stream. The PushStream created by this method is not fully asynchronous, it uses the connecting thread to consume the Java 8 Stream. As a result the streams created using this method will block terminal operations. This method should therefore not normally be used for infinite event streams, but instead for short, finite streams of data that can be processed rapidly, for example as the result of a flatmapping operation. In this scenario reusing the incoming thread improves performance. In the following example an incoming list of URLs is registered for download.

```
PushStreamProvider psp = new PushStreamProvider();

PushStream<List<URL>> urls = getURLStream();

urls.flatMap(l -> psp.streamOf(l.stream()))
    .forEach(url -> registerDownload(url));
```

For larger Streams of data, or when truly asynchronous operation is required, there is a second streamOf(Executor,ScheduledExecutorService,Stream) method which allows for asynchronous consumption of the stream. The Executor is used to consume elements from the Java 8 Stream using a single task. This mode of operation is suitable for use with infinite data streams, or for streams which require a truly asynchronous mode of operation, and does not require the stream to be parallel. If `null` is passed for the `Executor` then the PushStreamProvider will create a fixed thread pool of size 2. This allows for work to continue in the Push Stream even if the passed-in Stream blocks the consuming thread. If `null` is passed for the `ScheduledExecutor` then the Push Stream Provider may create a new scheduler or use a shared default.

706.5 Simple Push Event Sources

The PushEventSource and PushEventConsumer are both functional interfaces, however it is noticeably harder to implement a PushEventSource than a PushEventConsumer. A PushEventSource must be able to support multiple independently closeable consumer registrations, all of which are providing potentially different amounts of back pressure.

To simplify the case where a user wishes to write a basic event source the PushStreamProvider is able to create a SimplePushEventSource. The SimplePushEventSource handles the details of implementing PushEventSource, providing a simplified API for the event producing code to use.

Events can be sent via the Simple Push Event Source `publish(T)` method at any time until it is closed. These events may be silently ignored if no consumer is connected, but if one or more consumers are connected then the event will be asynchronously delivered to them.

Close or error events can be sent equally easily using the `endOfStream()` and `error(Throwable)` methods. These will send disconnection events to all of the currently connected consumers and remove them from the Simple Push Event Source. Note that sending these events does not close the Simple Push Event Source. Subsequent connection attempts will succeed, and events can still be published.

706.5.1 Optimizing Event Creation

In addition to the publication methods the Simple Push Event Source provides `isConnected()` and `connectPromise()` methods. The isConnected method gives a point-in-time snapshot of whether there are any connections to the Simple Push Event Source. If this method returns false then the event producer may wish to avoid creating the event, particularly if it is computationally expensive to do so. The connectPromise method returns a Promise representing the current connection state. This Promise resolves when there is a client connected (which means it may be resolved immediately as it is created). If the Simple Push Event Source is closed before the Promise resolves then the Promise is failed with an IllegalStateException. The connect Promise can be used to trigger the initialization of an event thread, allowing lazier startup.

```
PushStreamProvider psp = new PushStreamProvider();

SimplePushEventSource<Long> ses = psp.createSimpleEventSource(Long.class))

Success<Void,Void> onConnect = p -> {
    new Thread(() -> {
        long counter = 0;
        // Keep going as long as someone is listening
        while (ses.isConnected()) {
          ses.publish(++counter);
          Thread.sleep(100);
          System.out.println("Published: " + counter);
        }
        // Restart delivery when a new listener connects
        ses.connectPromise().then(onConnect);
    }).start();
   return null;
  };

// Begin delivery when someone is listening
ses.connectPromise().then(onConnect);

// Create a listener which prints out even numbers
psp.createStream(ses).
  filter(l -> l % 2L == 0).
  limit(5000L).
  forEach(f -> System.out.println("Consumed event: " + f));
```

706.6 Security

The Push Stream API does not define any OSGi services nor does the API perform any privileged actions. Therefore, it has no security considerations.

706.7 org.osgi.util.pushstream

Push Stream Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.pushstream; version="[1.1,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.util.pushstream; version="[1.1,1.2)"

706.7.1 Summary

- BufferBuilder - Create a buffered section of a Push-based stream
- PushbackPolicy - A PushbackPolicy is used to calculate how much back pressure to apply based on the current buffer.
- PushbackPolicyOption - PushbackPolicyOption provides a standard set of simple PushbackPolicy implementations.
- PushEvent - A PushEvent is an immutable object that is transferred through a communication channel to push information to a downstream consumer.
- PushEvent.EventType - The type of a PushEvent.
- PushEventConsumer - An Async Event Consumer asynchronously receives Data events until it receives either a Close or Error event.
- PushEventSource - An event source.
- PushStream - A Push Stream fulfills the same role as the Java 8 stream but it reverses the control direction.
- PushStreamBuilder - A Builder for a PushStream.
- PushStreamProvider - A factory for PushStream instances, and utility methods for handling PushEventSources and PushEventConsumers
- QueuePolicy - A QueuePolicy is used to control how events should be queued in the current buffer.

- QueuePolicyOption - QueuePolicyOption provides a standard set of simple QueuePolicy implementations.
- SimplePushEventSource - A SimplePushEventSource is a helper that makes it simpler to write a PushEventSource.
- ThresholdPushbackPolicy - Provides a configurable PushbackPolicy implementation that returns zero back pressure until the buffer fills beyond the supplied threshold.

706.7.2 public interface BufferBuilder<R, T, U extends BlockingQueue<PushEvent<? extends T>>>

<R> The type of object being built

<T> The type of objects in the PushEvent

<U> The type of the Queue used in the user specified buffer

Create a buffered section of a Push-based stream

Provider Type Consumers of this API must not implement this type

706.7.2.1 public R build()

Returns the object being built

706.7.2.2 public BufferBuilder<R, T, U> withBuffer(U queue)

queue

□ The BlockingQueue implementation to use as a buffer

Returns this builder

706.7.2.3 public BufferBuilder<R, T, U> withExecutor(Executor executor)

executor

□ Set the Executor that should be used to deliver events from this buffer

Returns this builder

706.7.2.4 public BufferBuilder<R, T, U> withParallelism(int parallelism)

parallelism

□ Set the maximum permitted number of concurrent event deliveries allowed from this buffer

Returns this builder

706.7.2.5 public BufferBuilder<R, T, U> withPushbackPolicy(PushbackPolicy<T, U> pushbackPolicy)

pushbackPolicy

□ Set the PushbackPolicy of this builder

Returns this builder

706.7.2.6 public BufferBuilder<R, T, U> withPushbackPolicy(PushbackPolicyOption pushbackPolicyOption, long time)

pushbackPolicyOption

time

□ Set the PushbackPolicy of this builder

Returns this builder

706.7.2.7 public BufferBuilder<R, T, U> withQueuePolicy(QueuePolicy<T, U> queuePolicy)

queuePolicy

□ Set the QueuePolicy of this Builder

Returns this builder

706.7.2.8 public BufferBuilder<R, T, U> withQueuePolicy(QueuePolicyOption queuePolicyOption)

queuePolicyOption

□ Set the QueuePolicy of this Builder

Returns this builder

706.7.2.9 public BufferBuilder<R, T, U> withScheduler(ScheduledExecutorService scheduler)

scheduler

□ Set the ScheduledExecutorService that should be used to trigger timed events after this buffer

Returns this builder

706.7.3 public interface PushbackPolicy<T, U extends BlockingQueue<PushEvent<? extends T>>>

<T> The type of the data

<U> The type of the queue

A PushbackPolicy is used to calculate how much back pressure to apply based on the current buffer. The PushbackPolicy will be called after an event has been queued, and the returned value will be used as back pressure.

See Also PushbackPolicyOption

706.7.3.1 public long pushback(U queue) throws Exception

queue

□ Given the current state of the queue, determine the level of back pressure that should be applied

Returns a back pressure value in nanoseconds

Throws Exception–

706.7.4 enum PushbackPolicyOption

PushbackPolicyOption provides a standard set of simple PushbackPolicy implementations.

See Also PushbackPolicy

706.7.4.1 FIXED

Returns a fixed amount of back pressure, independent of how full the buffer is

706.7.4.2 ON_FULL_FIXED

Returns zero back pressure until the buffer is full, then it returns a fixed value

706.7.4.3 ON_FULL_EXPONENTIAL

Returns zero back pressure until the buffer is full, then it returns an exponentially increasing amount, starting with the supplied value and doubling it each time. Once the buffer is no longer full the back pressure returns to zero.

706.7.4.4 LINEAR

Returns zero back pressure when the buffer is empty, then it returns a linearly increasing amount of back pressure based on how full the buffer is. The maximum value will be returned when the buffer is full.

706.7.4.5 public abstract PushbackPolicy<T, U> getPolicy(long value)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

value

□ Create a PushbackPolicy instance configured with a base back pressure time in nanoseconds The actual backpressure returned will vary based on the selected implementation, the base value, and the state of the buffer.

Returns A PushbackPolicy to use

706.7.4.6 public static PushbackPolicyOption valueOf(String name)

706.7.4.7 public static PushbackPolicyOption[] values()

706.7.5 public abstract class PushEvent<T>

<T> The payload type of the event.

A PushEvent is an immutable object that is transferred through a communication channel to push information to a downstream consumer. The event has three different types:

- EventType.DATA – Provides access to a typed data element in the stream.
- EventType.CLOSE – The stream is closed. After receiving this event, no more events will follow.
- EventType.ERROR – The stream ran into an unrecoverable problem and is sending the reason downstream. The stream is closed and no more events will follow after this event.

Concurrency Immutable

Provider Type Consumers of this API must not implement this type

706.7.5.1 public static PushEvent<T> close()

Type Parameters `<T>`

<T> The payload type.

□ Create a new close event.

Returns A new close event.

706.7.5.2 public static PushEvent<T> data(T payload)

Type Parameters `<T>`

<T> The payload type.

payload The payload.

□ Create a new data event.

Returns A new data event wrapping the specified payload.

706.7.5.3 public static PushEvent<T> error(Throwable t)

Type Parameters `<T>`

<T> The payload type.

t The error.

□ Create a new error event.

Returns A new error event with the specified error.

706.7.5.4 public T getData()

□ Return the data for this event.

Returns The data payload.

Throws `IllegalStateException`– if this event is not a EventType.DATA event.

706.7.5.5 public Throwable getFailure()

□ Return the error that terminated the stream.

Returns The error that terminated the stream.

Throws `IllegalStateException`– if this event is not an EventType.ERROR event.

706.7.5.6 public abstract PushEvent.EventType getType()

□ Get the type of this event.

Returns The type of this event.

706.7.5.7 public boolean isTerminal()

□ Answer if no more events will follow after this event.

Returns `false` if this is a data event, otherwise `true`.

706.7.5.8 public PushEvent<X> nodata()

Type Parameters `<X>`

<X> The new payload type.

□ Convenience to cast a close/error event to another payload type. Since the payload type is not needed for these events this is harmless. This therefore allows you to forward the close/error event downstream without creating anew event.

Returns The current error or close event mapped to a new payload type.

Throws `IllegalStateException`– if the event is a EventType.DATA event.

706.7.6 enum PushEvent.EventType

The type of a PushEvent.

706.7.6.1 DATA

A data event forming part of the stream

706.7.6.2 ERROR

An error event that indicates streaming has failed and that no more events will arrive

706.7.6.3 CLOSE

An event that indicates that the stream has terminated normally

706.7.6.4 public static PushEvent.EventType valueOf(String name)

706.7.6.5 public static PushEvent.EventType[] values()

706.7.7 public interface PushEventConsumer<T>

<T> The type for the event payload

An Async Event Consumer asynchronously receives Data events until it receives either a Close or Error event.

706.7.7.1 public static final long ABORT = -1L

If ABORT is used as return value, the sender should close the channel all the way to the upstream source. The ABORT will not guarantee that no more events are delivered since this is impossible in a concurrent environment. The consumer should accept subsequent events and close/clean up when the Close or Error event is received. Though ABORT has the value -1, any value less than 0 will act as an abort.

706.7.7.2 public static final long CONTINUE = 0L

A 0 indicates that the consumer is willing to receive subsequent events at full speeds. Any value more than 0 will indicate that the consumer is becoming overloaded and wants a delay of the given milliseconds before the next event is sent. This allows the consumer to pushback the event delivery speed.

706.7.7.3 public long accept(PushEvent<? extends T> event) throws Exception

event The event

□ Accept an event from a source. Events can be delivered on multiple threads simultaneously. However, Close and Error events are the last events received, no more events must be sent after them.

Returns less than 0 means abort, 0 means continue, more than 0 means delay ms

Throws `Exception` – to indicate that an error has occurred and that no further events should be delivered to this PushEventConsumer

706.7.8 public interface PushEventSource<T>

<T> The payload type

An event source. An event source can open a channel between a source and a consumer. Once the channel is opened (even before it returns) the source can send events to the consumer. A source should stop sending and automatically close the channel when sending an event returns a negative value, see PushEventConsumer.ABORT. Values that are larger than 0 should be treated as a request to delay the next events with those number of milliseconds.

706.7.8.1 public AutoCloseable open(PushEventConsumer<? super T> aec) throws Exception

aec the consumer (not null)

□ Open the asynchronous channel between the source and the consumer. The call returns an AutoCloseable. This can be closed, and should close the channel, including sending a Close event if the channel was not already closed. The returned object must be able to be closed multiple times without sending more than one Close events.

Returns a AutoCloseable that can be used to close the stream

Throws `Exception` –

706.7.9 public interface PushStream<T> extends AutoCloseable

<T> The Payload type

A Push Stream fulfills the same role as the Java 8 stream but it reverses the control direction. The Java 8 stream is pull based and this is push based. A Push Stream makes it possible to build a pipeline of transformations using a builder kind of model. Just like streams, it provides a number of terminating methods that will actually open the channel and perform the processing until the channel is closed (The source sends a Close event). The results of the processing will be send to a Promise, just like any error events. A stream can be used multiple times. The Push Stream represents a pipeline. Upstream is in the direction of the source, downstream is in the direction of the terminating method. Events are sent downstream asynchronously with no guarantee for ordering or concurrency. Methods are available to provide serialization of the events and splitting in background threads.

Provider Type Consumers of this API must not implement this type

706.7.9.1 public PushStream<T> adjustBackPressure(LongUnaryOperator adjustment)

adjustment

□ Changes the back-pressure propagated by this pipeline stage.

The supplied function receives the back pressure returned by the next pipeline stage and returns the back pressure that should be returned by this stage. This function will not be called if the previous pipeline stage returns negative back pressure.

Returns Builder style (can be a new or the same object)

706.7.9.2 public PushStream<T> adjustBackPressure(ToLongBiFunction<T, Long> adjustment)

adjustment

□ Changes the back-pressure propagated by this pipeline stage.

The supplied function receives the data object passed to the next pipeline stage and the back pressure that was returned by that stage when accepting it. The function returns the back pressure that should be returned by this stage. This function will not be called if the previous pipeline stage returns negative back pressure.

Returns Builder style (can be a new or the same object)

706.7.9.3 public Promise<Boolean> allMatch(Predicate<? super T> predicate)

predicate

□ Closes the channel and resolve the promise with false when the predicate does not matches a pay load. If the channel is closed before, the promise is resolved with true.

This is a **short circuiting terminal operation**

Returns A Promise that will resolve when an event fails to match the predicate, or the end of the stream is reached

706.7.9.4 public Promise<Boolean> anyMatch(Predicate<? super T> predicate)

predicate

□ Close the channel and resolve the promise with true when the predicate matches a payload. If the channel is closed before the predicate matches, the promise is resolved with false.

This is a **short circuiting terminal operation**

Returns A Promise that will resolve when an event matches the predicate, or the end of the stream is reached

706.7.9.5 public PushStream<R> asyncMap(int n, int delay, Function<? super T, Promise<? extends R>> mapper)

Type Parameters <R>

n number of simultaneous promises to use

delay Nr of ms/promise that is queued back pressure

mapper The mapping function

□ Asynchronously map the payload values. The mapping function returns a Promise representing the asynchronous mapping operation.

The PushStream limits the number of concurrently running mapping operations, and returns back pressure based on the number of existing queued operations.

Returns Builder style (can be a new or the same object)

Throws `IllegalArgumentException` – if the number of threads is < 1 or the delay is < 0

`NullPointerException` – if the mapper is null

706.7.9.6 public PushStream<T> buffer()

□ Buffer the events in a queue using default values for the queue size and other behaviors. Buffered work will be processed asynchronously in the rest of the chain. Buffering also blocks the transmission of back pressure to previous elements in the chain, although back pressure is honored by the buffer.

Buffers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream event consumers. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed. For fast sources filter(Predicate) and coalesce(int, Function) fork(int, int, Executor) are better choices.

Returns Builder style (can be a new or the same object)

706.7.9.7 public PushStreamBuilder<T, U> buildBuffer()

Type Parameters `<U extends BlockingQueue<PushEvent<? extends T>>>`

□ Build a buffer to enqueue events in a queue using custom values for the queue size and other behaviors. Buffered work will be processed asynchronously in the rest of the chain. Buffering also blocks the transmission of back pressure to previous elements in the chain, although back pressure is honored by the buffer.

Buffers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream event consumers. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed. For fast sources filter(Predicate) and coalesce(int, Function) fork(int, int, Executor) are better choices.

Buffers are also useful as "circuit breakers" in the pipeline. If a QueuePolicyOption.FAIL is used then a full buffer will trigger the stream to close, preventing an event storm from reaching the client.

Returns A builder which can be used to configure the buffer for this pipeline stage.

706.7.9.8 public void close()

□ Close this PushStream by sending an event of type PushEvent.EventType.CLOSE downstream. Closing a PushStream is a safe operation that will not throw an Exception.

Calling `close()` on a closed PushStream has no effect.

706.7.9.9 public PushStream<R> coalesce(Function<? super T, Optional<R>> f)

Type Parameters `<R>`

f

□ Coalesces a number of events into a new type of event. The input events are forwarded to a accumulator function. This function returns an Optional. If the optional is present, it's value is send downstream, otherwise it is ignored.

Returns Builder style (can be a new or the same object)

706.7.9.10 public PushStream<R> coalesce(int count, Function<Collection<T>, R> f)

Type Parameters `<R>`

count

f

□ Coalesces a number of events into a new type of event. A fixed number of input events are forwarded to a accumulator function. This function returns new event data to be forwarded on.

Returns Builder style (can be a new or the same object)

706.7.9.11 public PushStream<R> coalesce(IntSupplier count, Function<Collection<T>, R> f)

Type Parameters <R>

count

f

□ Coalesces a number of events into a new type of event. A variable number of input events are forwarded to a accumulator function. The number of events to be forwarded is determined by calling the count function. The accumulator function then returns new event data to be forwarded on.

Returns Builder style (can be a new or the same object)

706.7.9.12 public Promise<R> collect(Collector<? super T, A, R> collector)

Type Parameters <R, A>

collector

□ See Stream. Will resolve once the channel closes.

This is a **terminal operation**

Returns A Promise representing the collected results

706.7.9.13 public Promise<Long> count()

□ See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

Returns A Promise representing the number of values in the stream

706.7.9.14 public PushStream<T> distinct()

□ Remove any duplicates. Notice that this can be expensive in a large stream since it must track previous payloads.

Returns Builder style (can be a new or the same object)

706.7.9.15 public PushStream<T> filter(Predicate<? super T> predicate)

predicate The predicate that is tested (not null)

□ Only pass events downstream when the predicate tests true.

Returns Builder style (can be a new or the same object)

706.7.9.16 public Promise<Optional<T>> findAny()

□ Close the channel and resolve the promise with the first element. If the channel is closed before, the Optional will have no value.

This is a **terminal operation**

Returns a promise

706.7.9.17 public Promise<Optional<T>> findFirst()

□ Close the channel and resolve the promise with the first element. If the channel is closed before, the Optional will have no value.

Returns a promise

706.7.9.18 public PushStream<R> flatMap(Function<? super T, ? extends PushStream<? extends R>> mapper)

Type Parameters <R>

mapper The flat map function

□ Flat map the payload value (turn one event into 0..n events of potentially another type).

Returns Builder style (can be a new or the same object)

706.7.9.19 public Promise<Void> forEach(Consumer<? super T> action)

action The action to perform

□ Execute the action for each event received until the channel is closed. This is a terminating method, the returned promise is resolved when the channel closes.

This is a **terminal operation**

Returns A promise that is resolved when the channel closes.

706.7.9.20 public Promise<Long> forEachEvent(PushEventConsumer<? super T> action)

action

□ Pass on each event to another consumer until the stream is closed.

This is a **terminal operation**

Returns a promise

706.7.9.21 public PushStream<T> fork(int n, int delay, Executor e)

n number of simultaneous background threads to use

delay Nr of ms/thread that is queued back pressure

e an executor to use for the background threads.

□ Execute the downstream events in up to n background threads. If more requests are outstanding apply delay * nr of delayed threads back pressure. A downstream channel that is closed or throws an exception will cause all execution to cease and the stream to close

Returns Builder style (can be a new or the same object)

Throws `IllegalArgumentException` – if the number of threads is < 1 or the delay is < 0

`NullPointerException` – if the Executor is null

706.7.9.22 public PushStream<T> limit(long maxSize)

maxSize Maximum number of elements has been received

□ Automatically close the channel after the maxSize number of elements is received.

Returns Builder style (can be a new or the same object)

706.7.9.23 public PushStream<T> limit(Duration maxTime)

maxTime The maximum time that the stream should remain open

□ Automatically close the channel after the given amount of time has elapsed.

Returns Builder style (can be a new or the same object)

706.7.9.24 public PushStream<R> map(Function<? super T, ? extends R> mapper)

Type Parameters <R>

mapper The map function

□ Map a payload value.

Returns Builder style (can be a new or the same object)

706.7.9.25 public Promise<Optional<T>> max(Comparator<? super T> comparator)

comparator

□ See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

Returns A Promise representing the maximum value, or null if no values are seen before the end of the stream

706.7.9.26 public PushStream<T> merge(PushEventSource<? extends T> source)

source The source to merge in.

□ Merge in the events from another source. The resulting channel is not closed until this channel and the channel from the source are closed.

Returns Builder style (can be a new or the same object)

706.7.9.27 public PushStream<T> merge(PushStream<? extends T> source)

source The source to merge in.

□ Merge in the events from another PushStream. The resulting channel is not closed until this channel and the channel from the source are closed.

Returns Builder style (can be a new or the same object)

706.7.9.28 public Promise<Optional<T>> min(Comparator<? super T> comparator)

comparator

□ See Stream. Will resolve onces the channel closes.

This is a **terminal operation**

Returns A Promise representing the minimum value, or null if no values are seen before the end of the stream

706.7.9.29 public Promise<Boolean> noneMatch(Predicate<? super T> predicate)

predicate

□ Closes the channel and resolve the promise with false when the predicate matches any pay load. If the channel is closed before, the promise is resolved with true.

This is a **short circuiting terminal operation**

Returns A Promise that will resolve when an event matches the predicate, or the end of the stream is reached

706.7.9.30 public PushStream<T> onClose(Runnable closeHandler)

closeHandler Will be called on close

□ Provide a handler that must be run after the PushStream is closed. This handler will run after any downstream operations have processed the terminal event but before any upstream operations have processed the terminal event.

Returns This stream

706.7.9.31 public PushStream<T> onError(Consumer<? super Throwable> errorHandler)

errorHandler Will be called on an error event

□ Provide a handler that will be called if the PushStream is closed with an event of type PushEvent.EventType.ERROR. The error value from this event will be passed to the callback function after the PushStream is closed. This handler will run after any downstream operations have processed the error event but before any upstream operations have processed the error event.

Returns This stream

706.7.9.32 public Promise<T> reduce(T identity, BinaryOperator<T> accumulator)

identity The identity/begin value

accumulator The accumulator

□ Standard reduce, see Stream. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

Returns A

706.7.9.33 public Promise<Optional<T>> reduce(BinaryOperator<T> accumulator)

accumulator The accumulator

□ Standard reduce without identity, so the return is an Optional. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

Returns an Optional

706.7.9.34 public Promise<U> reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

Type Parameters <U>

identity

accumulator

combiner combines two U's into one U (for example, combine two lists)

□ Standard reduce with identity, accumulator and combiner. The returned promise will be resolved when the channel closes.

This is a **terminal operation**

Returns The promise

706.7.9.35 public PushStream<T> sequential()

□ Ensure that any events are delivered sequentially. That is, no overlapping calls downstream. This can be used to turn a forked stream (where for example a heavy conversion is done in multiple threads) back into a sequential stream so a reduce is simple to do.

Returns Builder style (can be a new or the same object)

706.7.9.36 public PushStream<T> skip(long n)

n number of elements to skip

□ Skip a number of events in the channel.

Returns Builder style (can be a new or the same object)

Throws `IllegalArgumentException` – if the number of events to skip is negative

706.7.9.37 public PushStream<T> sorted()

□ Sorted the elements, assuming that T extends Comparable. This is of course expensive for large or infinite streams since it requires buffering the stream until close.

Returns Builder style (can be a new or the same object)

706.7.9.38 public PushStream<T> sorted(Comparator<? super T> comparator)

comparator

□ Sorted the elements with the given comparator. This is of course expensive for large or infinite streams since it requires buffering the stream until close.

Returns Builder style (can be a new or the same object)

706.7.9.39 public PushStream<T>[] split(Predicate<? super T>... predicates)

predicates the predicates to test

□ Split the events to different streams based on a predicate. If the predicate is true, the event is dispatched to that channel on the same position. All predicates are tested for every event.

This method differs from other methods of PushStream in three significant ways:

- The return value contains multiple streams.
- This stream will only close when all of these child streams have closed.
- Event delivery is made to all open children that accept the event.

Returns streams that map to the predicates

706.7.9.40 public PushStream<T> timeout(Duration idleTime)

idleTime The length of time that the stream should remain open when no events are being received.

□ Automatically fail the channel if no events are received for the indicated length of time. If the timeout is reached then a failure event containing a TimeoutException will be sent.

Returns Builder style (can be a new or the same object)

706.7.9.41 public Promise<Object> toArray()

□ Collect the payloads in an Object array after the channel is closed. This is a terminating method, the returned promise is resolved when the channel is closed.

This is a **terminal operation**

Returns A promise that is resolved with all the payloads received over the channel

706.7.9.42 public Promise<A> toArray(IntFunction<A> generator)

Type Parameters <A>

<A> The element type of the resulting array.

generator A function which returns an array into which the payloads are stored.

□ Collect the payloads in an Object array after the channel is closed. This is a terminating method, the returned promise is resolved when the channel is closed. The type of the array is handled by the caller using a generator function that gets the length of the desired array.

This is a **terminal operation**

Returns A promise that is resolved with all the payloads received over the channel. The promise will be failed with an ArrayStoreException if the runtime type of the array returned by the array generator is not a supertype of the runtime type of every payload in the channel.

706.7.9.43 public PushStream<R> window(Duration d, Function<Collection<T>, R> f)

Type Parameters <R>

d

f

□ Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. This function returns new event data to be forwarded on. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this PushStream.
- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages

Returns Builder style (can be a new or the same object)

706.7.9.44 public PushStream<R> window(Duration d, Executor executor, Function<Collection<T>, R> f)

Type Parameters <R>

d

executor

f

□ Buffers a number of events over a fixed time interval and then forwards the events to an accumulator function. This function returns new event data to be forwarded on. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered by a task given to the supplied executor.
- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages

Returns Builder style (can be a new or the same object)

706.7.9.45 public PushStream<R> window(Supplier<Duration> timeSupplier, IntSupplier maxEvents, BiFunction<Long, Collection<T>, R> f)

Type Parameters <R>

timeSupplier

maxEvents

f

□ Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. The length of time over which events are buffered is determined by the time function. A maximum number of events can also be requested, if this number of events is reached then the accumulator will be called early. The accumulator function returns new event data to be forwarded on. It is also given the length of time for which the buffer accumulated data. This may be less than the requested interval if the buffer reached the maximum number of requested events early. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this PushStream.

- Due to the buffering and asynchronous delivery required, this method prevents the propagation of back-pressure to earlier stages
- If the window finishes by hitting the maximum number of events then the remaining time in the window will be applied as back-pressure to the previous stage, attempting to slow the producer to the expected windowing threshold.

Returns Builder style (can be a new or the same object)

706.7.9.46 public PushStream<R> window(Supplier<Duration> timeSupplier, IntSupplier maxEvents, Executor executor, BiFunction<Long, Collection<T>, R> f)

Type Parameters <R>

timeSupplier

maxEvents

executor

f

□ Buffers a number of events over a variable time interval and then forwards the events to an accumulator function. The length of time over which events are buffered is determined by the time function. A maximum number of events can also be requested, if this number of events is reached then the accumulator will be called early. The accumulator function returns new event data to be forwarded on. It is also given the length of time for which the buffer accumulated data. This may be less than the requested interval if the buffer reached the maximum number of requested events early. Note that:

- The collection forwarded to the accumulator function will be empty if no events arrived during the time interval.
- The accumulator function will be run and the forwarded event delivered as a different task, (and therefore potentially on a different thread) from the one that delivered the event to this PushStream.
- If the window finishes by hitting the maximum number of events then the remaining time in the window will be applied as back-pressure to the previous stage, attempting to slow the producer to the expected windowing threshold.

Returns Builder style (can be a new or the same object)

706.7.10 public interface PushStreamBuilder<T, U extends BlockingQueue<PushEvent<? extends T>>> extends BufferBuilder<PushStream<T>, T, U>

<T> The type of objects in the PushEvent

<U> The type of the Queue used in the user specified buffer

A Builder for a PushStream. This Builder extends the support of a standard BufferBuilder by allowing the PushStream to be unbuffered.

Provider Type Consumers of this API must not implement this type

706.7.10.1 public PushStreamBuilder<T, U> unbuffered()

□ Tells this PushStreamBuilder to create an unbuffered stream which delivers events directly to its consumer using the incoming delivery thread. Setting the PushStreamBuilder to be unbuffered means that any buffer, queue policy or push back policy will be ignored. Note that calling one of:

- withBuffer(BlockingQueue)
- withQueuePolicy(QueuePolicy)

- withQueuePolicy(QueuePolicyOption)
- withPushbackPolicy(PushbackPolicy)
- withPushbackPolicy(PushbackPolicyOption, long)
- withParallelism(int)

after this method will reset this builder to require a buffer.

Returns the builder

706.7.10.2 public PushStreamBuilder<T, U> withBuffer(U queue)

queue

□ The BlockingQueue implementation to use as a buffer

Returns this builder

706.7.10.3 public PushStreamBuilder<T, U> withExecutor(Executor executor)

executor

□ Set the Executor that should be used to deliver events from this buffer

Returns this builder

706.7.10.4 public PushStreamBuilder<T, U> withParallelism(int parallelism)

parallelism

□ Set the maximum permitted number of concurrent event deliveries allowed from this buffer

Returns this builder

706.7.10.5 public PushStreamBuilder<T, U> withPushbackPolicy(PushbackPolicy<T, U> pushbackPolicy)

pushbackPolicy

□ Set the PushbackPolicy of this builder

Returns this builder

706.7.10.6 public PushStreamBuilder<T, U> withPushbackPolicy(PushbackPolicyOption pushbackPolicyOption, long time)

pushbackPolicyOption

time

□ Set the PushbackPolicy of this builder

Returns this builder

706.7.10.7 public PushStreamBuilder<T, U> withQueuePolicy(QueuePolicy<T, U> queuePolicy)

queuePolicy

□ Set the QueuePolicy of this Builder

Returns this builder

706.7.10.8 public PushStreamBuilder<T, U> withQueuePolicy(QueuePolicyOption queuePolicyOption)

queuePolicyOption

□ Set the QueuePolicy of this Builder

Returns this builder

706.7.10.9 public PushStreamBuilder<T, U> withScheduler(ScheduledExecutorService scheduler)

scheduler

□ Set the ScheduledExecutorService that should be used to trigger timed events after this buffer

Returns this builder

706.7.11 public final class PushStreamProvider

A factory for PushStream instances, and utility methods for handling PushEventSources and PushEventConsumers

706.7.11.1 public PushStreamProvider()

706.7.11.2 public BufferBuilder<PushEventConsumer<T>, T, U> buildBufferedConsumer(PushEventConsumer<T> delegate)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

delegate

□ Build a buffered PushEventConsumer with custom configuration.

The returned consumer will be buffered from the event source, and will honor back pressure requests from its delegate even if the event source does not.

Buffered consumers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm the consumer. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

Buffers are also useful as "circuit breakers". If a QueuePolicyOption.FAIL is used then a full buffer will request that the stream close, preventing an event storm from reaching the client.

Note that this buffered consumer will close when it receives a terminal event, or if the delegate returns negative backpressure. No further events will be propagated after this time.

Returns a PushEventConsumer with a buffer directly before it

706.7.11.3 public BufferBuilder<PushEventSource<T>, T, U> buildEventSourceFromStream(PushStream<T> stream)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

stream

□ Convert an PushStream into an PushEventSource. The first call to PushEventSource.open(PushEventConsumer) will begin event processing.

The PushEventSource will remain active until the backing stream is closed, and permits multiple consumers to PushEventSource.open(PushEventConsumer) it. Note that this means the caller of this method is responsible for closing the supplied stream if it is not finite in length.

Late joining consumers will not receive historical events, but will immediately receive the terminal event which closed the stream if the stream is already closed.

Returns a PushEventSource backed by the PushStream

706.7.11.4 public BufferBuilder<SimplePushEventSource<T>, T, U> buildSimpleEventSource(Class<T> type)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

type

□ Build a SimplePushEventSource with the supplied type and custom buffering behaviors. The SimplePushEventSource will respond to back pressure requests from the consumers connected to it.

Returns a SimplePushEventSource

706.7.11.5 public PushStreamBuilder<T, U> buildStream(PushEventSource<T> eventSource)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

eventSource The source of the events

□ Builds a push stream with custom configuration.

The resulting PushStream may be buffered or unbuffered depending on how it is configured.

Returns A PushStreamBuilder for the stream

706.7.11.6 public PushEventConsumer<T> createBufferedConsumer(PushEventConsumer<T> delegate)

Type Parameters `<T>`

delegate

□ Create a buffered PushEventConsumer with the default configured buffer, executor size, queue, queue policy and pushback policy. This is equivalent to calling

```
buildBufferedConsumer(delegate).create();
```

The returned consumer will be buffered from the event source, and will honor back pressure requests from its delegate even if the event source does not.

Buffered consumers are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm the consumer. Buffering will not, however, protect downstream components from a source which produces events faster than they can be consumed.

Returns a PushEventConsumer with a buffer directly before it

706.7.11.7 public PushEventSource<T> createEventSourceFromStream(PushStream<T> stream)

Type Parameters `<T>`

stream

□ Convert an PushStream into an PushEventSource. The first call to PushEventSource.open(PushEventConsumer) will begin event processing. The PushEventSource will remain active until the backing stream is closed, and permits multiple consumers to PushEventSource.open(PushEventConsumer) it. This is equivalent to:

```
buildEventSourceFromStream(stream).create();
```

Returns a PushEventSource backed by the PushStream

706.7.11.8 public SimplePushEventSource<T> createSimpleEventSource(Class<T> type)

Type Parameters `<T>`

type

□ Create a SimplePushEventSource with the supplied type and default buffering behaviors. The SimplePushEventSource will respond to back pressure requests from the consumers connected to it. This is equivalent to:

```
buildSimpleEventSource(type).create();
```

Returns a SimplePushEventSource

706.7.11.9 public PushStream<T> createStream(PushEventSource<T> eventSource)

Type Parameters <T>

eventSource

□ Create a stream with the default configured buffer, executor size, queue, queue policy and pushback policy. This is equivalent to calling

```
 buildStream(source).create();
```

This stream will be buffered from the event producer, and will honor back pressure even if the source does not.

Buffered streams are useful for "bursty" event sources which produce a number of events close together, then none for some time. These bursts can sometimes overwhelm downstream processors. Buffering will not, however, protect downstream components from a source which produces events faster (on average) than they can be consumed.

Event delivery will not begin until a terminal operation is reached on the chain of PushStreams. Once a terminal operation is reached the stream will be connected to the event source.

Returns A PushStream with a default initial buffer

706.7.11.10 public PushStream<T> streamOf(Stream<T> items)

Type Parameters <T>

items The items to push into the PushStream

□ Create an Unbuffered PushStream from a Java Stream The data from the stream will be pushed into the PushStream synchronously as it is opened. This may make terminal operations blocking unless a buffer has been added to the PushStream. Care should be taken with infinite Streams to avoid blocking indefinitely.

Returns A PushStream containing the items from the Java Stream

706.7.11.11 public PushStream<T> streamOf(Executor executor, ScheduledExecutorService scheduler, Stream<T> items)

Type Parameters <T>

executor The worker to use to push items from the Stream into the PushStream

scheduler The scheduler to use to trigger timed events in the PushStream

items The items to push into the PushStream

□ Create an Unbuffered PushStream from a Java Stream The data from the stream will be pushed into the PushStream asynchronously using the supplied Executor.

Returns A PushStream containing the items from the Java Stream

706.7.12 public interface QueuePolicy<T, U extends BlockingQueue<PushEvent<? extends T>>>

<T> The type of the data

<U> The type of the queue

A QueuePolicy is used to control how events should be queued in the current buffer. The QueuePolicy will be called when an event has arrived.

See Also QueuePolicyOption

706.7.12.1 public void doOffer(U queue, PushEvent<? extends T> event) throws Exception

queue

event

□ Enqueue the event and return the remaining capacity available for events

Throws `Exception`– If an error occurred adding the event to the queue. This exception will cause the connection between the PushEventSource and the PushEventConsumer to be closed with an EventType.ERROR

706.7.13 enum QueuePolicyOption

QueuePolicyOption provides a standard set of simple QueuePolicy implementations.

See Also QueuePolicy

706.7.13.1 DISCARD_OLDEST

Attempt to add the supplied event to the queue. If the queue is unable to immediately accept the value then discard the value at the head of the queue and try again. Repeat this process until the event is enqueued.

706.7.13.2 BLOCK

Attempt to add the supplied event to the queue, blocking until the enqueue is successful.

706.7.13.3 FAIL

Attempt to add the supplied event to the queue, throwing an exception if the queue is full.

706.7.13.4 public abstract QueuePolicy<T, U> getPolicy()

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

Returns a QueuePolicy implementation

706.7.13.5 public static QueuePolicyOption valueOf(String name)

706.7.13.6 public static QueuePolicyOption[] values()

706.7.14 public interface SimplePushEventSource<T> extends PushEventSource<T>, AutoCloseable

<T> The type of the events produced by this source

A SimplePushEventSource is a helper that makes it simpler to write a PushEventSource. Users do not need to manage multiple registrations to the stream, nor do they have to be concerned with back pressure.

Provider Type Consumers of this API must not implement this type

706.7.14.1 public void close()

□ Close this source. Calling this method indicates that there will never be any more events published by it. Calling this method sends a close event to all connected consumers. After calling this method any PushEventConsumer that tries to open(PushEventConsumer) this source will immediately receive a close event, and will not see any remaining buffered events.

706.7.14.2 public Promise<Void> connectPromise()

□ This method can be used to delay event generation until an event source has connected. The returned promise will resolve as soon as one or more PushEventConsumer instances have opened the SimplePushEventSource.

The returned promise may already be resolved if this SimplePushEventSource already has connected consumers. If the SimplePushEventSource is closed before the returned Promise resolves then it will be failed with an IllegalStateException.

Note that the connected consumers are able to asynchronously close their connections to this SimplePushEventSource, and therefore it is possible that once the promise resolves this SimplePushEventSource may no longer be connected to any consumers.

Returns A promise representing the connection state of this EventSource

706.7.14.3 public void endOfStream()

□ Close this source for now, but potentially reopen it later. Calling this method asynchronously sends a close event to all connected consumers and then disconnects them. Any events previously queued by the publish(Object) method will be delivered before this close event.

After calling this method any PushEventConsumer that wishes may open(PushEventConsumer) this source, and will receive subsequent events.

706.7.14.4 public void error(Throwable t)

t the error

□ Close this source for now, but potentially reopen it later. Calling this method asynchronously sends an error event to all connected consumers and then disconnects them. Any events previously queued by the publish(Object) method will be delivered before this error event.

After calling this method any PushEventConsumer that wishes may open(PushEventConsumer) this source, and will receive subsequent events.

706.7.14.5 public boolean isConnected()

□ Determine whether there are any PushEventConsumers for this PushEventSource. This can be used to skip expensive event creation logic when there are no listeners.

Returns true if any consumers are currently connected

706.7.14.6 public void publish(T t)

t

□ Asynchronously publish an event to this stream and all connected PushEventConsumer instances. When this method returns there is no guarantee that all consumers have been notified. Events published by a single thread will maintain their relative ordering, however they may be interleaved with events from other threads.

Throws `IllegalStateException` – if the source is closed

706.7.15 public final class ThresholdPushbackPolicy<T, U extends BlockingQueue<PushEvent<? extends T>>> implements PushbackPolicy<T, U>

<T> The type of objects in the PushEvent

<U> The type of the Queue used in the user specified buffer

Provides a configurable PushbackPolicy implementation that returns zero back pressure until the buffer fills beyond the supplied threshold. Once the threshold is reached back pressure is returned based on the supplied parameters.

The starting level of the back pressure once the threshold is exceeded is defined using the `initial` parameter. Additional back pressure is applied for each queued item over the threshold. The amount of this additional back pressure is defined by the `increment` parameter.

The following common use cases are supported:

- The increment size can be zero, returning the initial value as a fixed back pressure after the threshold is exceeded.
- The initial value can be zero, returning a linearly increasing back pressure from zero once the threshold is exceeded

Since 1.1

706.7.15.1 public static ThresholdPushbackPolicy<T, U> createIncrementalThresholdPushbackPolicy(int threshold, long increment)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

increment the increments in which the pressure increases

threshold the queue size limit after which back pressure will be applied

□ A simple configuration, where the increment size is used as the initial back pressure.

Returns a new ThresholdPushbackPolicy

Throws `IllegalArgumentException`– if any of the given values is lower then zero

See Also createThresholdPushbackPolicy(int threshold, long initial, long increment)

706.7.15.2 public static ThresholdPushbackPolicy<T, U> createSimpleThresholdPushbackPolicy(int threshold)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

threshold the queue size limit after which back pressure will be applied

□ A simple configuration with an initial back pressure of one and and increase increment size of one.

Returns a new ThresholdPushbackPolicy

Throws `IllegalArgumentException`– if any of the given values is lower then zero

See Also createThresholdPushbackPolicy(int threshold, long initial, long increment)

706.7.15.3 public static ThresholdPushbackPolicy<T, U> createThresholdPushbackPolicy(int threshold, long initial, long increment)

Type Parameters `<T, U extends BlockingQueue<PushEvent<? extends T>>>`

increment the increments in which the pressure increases.

initial the initial back pressure which defines the floor after the threshold is exceeded

threshold the queue size limit after which back pressure will be applied

□ Provides a ThresholdPushbackPolicy with an individual configuration for all possible parameters.

Returns a new ThresholdPushbackPolicy

Throws `IllegalArgumentException`– if any of the given values is lower then zero

706.7.15.4 public long pushback(U queue) throws Exception

queue

□ Given the current state of the queue, determine the level of back pressure that should be applied

Returns a back pressure value in nanoseconds

Throws `Exception`–

706.8 References

[1] *Java 8 Stream API*

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#package.description

706.9 Changes

- Added new push back policy implementation ThresholdPushbackPolicy.

707 Converter Specification

Version 1.0

707.1 Introduction

Data conversion is an inherent part of writing software in a type safe language. In Java, converting strings to proper types or to convert one type to a more convenient type is often done manually. Any errors are then handled inline.

In release 6, the OSGi specifications introduced Data Transfer Objects (DTOs). DTOs are public objects without open generics that only contain public instance fields based on simple types, arrays, and collections. In many ways DTOs can be used as an alternative to Java beans. Java beans are hiding their fields and provide access methods which separates the contract (the public interface) from the internal usage. Though this model has advantages in technical applications it tends to add overhead. DTOs unify the specification with the data since the data is what is already public when it is sent to another process or serialized.

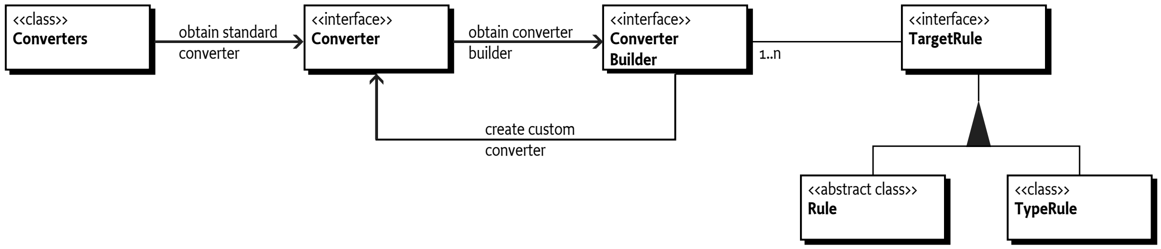
This specification defines the OSGi Converter that makes it easy to convert many types to other types, including scalars, Collections, Maps, Beans, Interfaces and DTOs without having to write the boilerplate conversion code. The converter strictly adheres to the rules specified in this chapter. Converters can also be customized using converter builders.

707.2 Entities

The following entities are used in this specification:

- *Converter* - a converter can perform conversion operations.
- *Standard Converter* - a converter implementation that follows this specification.
- *Converter Builder* - can create customized converters by specifying rules for specific conversions.
- *Source* - the object to be converted.
- *Target* - the target of the conversion.
- *Source Type* - the type of the source to be converted.
- *Target Type* - the desired type of the conversion target.
- *Rule* - a rule is used to customize the behavior of the converter.

Figure 707.1 Converter Entity overview



707.3 Standard Converter

The Standard Converter is a converter that follows precisely what is described in this specification. It converts source objects to the desired target type if a suitable conversion is available. An instance can be obtained by calling the static standardConverter() method on the Converters class.

Some example conversions:

```
Converter c = Converters.standardConverter();

// Scalar conversions
MyEnum e = c.convert(MyOtherEnum.BLUE).to(MyEnum.class);
BigDecimal bd = c.convert(12345).to(BigDecimal.class);

// Collection/array conversions
List<String> ls = Arrays.asList("978", "142", "-99");
long[] la = c.convert(ls).to(long[].class);

// Map conversions
Map someMap = new HashMap();
someMap.put("timeout", "700");
MyInterface mi = c.convert(someMap).to(MyInterface.class);
int t = mi.timeout(); // t=700
```

707.4 Conversions

For scalars, conversions are only performed when the target type is not compatible with the source type. For example, when requesting to convert a java.math.BigDecimal to a java.lang.Number the big decimal is simply used as-is as this type is assignable to the requested target type.

In the case of arrays, Collections and Map-like structures a new object is always returned, even if the target type is compatible with the source type. This copy can be owned and optionally further modified by the caller.

707.4.1 Generics

When converting to a target type with generic type parameters it is necessary to capture these to instruct the converter to produce the correct parameterized type. This can be achieved with the TypeReference based APIs, for example:

```
Converter c = Converters.standardConverter();
List<Long> list = c.convert("123").to(new TypeReference<List<Long>>(){});
// list will contain the Long value 123L
```

707.4.2 Scalars

707.4.2.1 Direct conversion between scalars

Direct conversion between the following scalars is supported:

Table 707.1 Scalar types that support direct conversions

to \ from	**Boolean**	**Character**	***Number***	null
boolean	v.booleanValue()	v.charValue() != 0	v.*number*Value() != 0	false
char	v.booleanValue() ? 1 : 0	v.charValue()	(char) v.intValue()	0

to \ from	**Boolean**	**Character**	***Number***	null
number	v.booleanValue() ? 1 : 0	(*number*) v.charValue()	v.*number*Value()	0

Where conversion is done from corresponding primitive types, these types are boxed before converting. Where conversion is done to corresponding boxed types, the types are boxed after converting.

Direct conversions between Enums and ints and between Dates and longs are also supported, see the sections below.

Conversions between from Map.Entry to scalars follow special rules, see *Map.Entry* on page 1460.

All other conversions between scalars are done by converting the source object to a String first and then converting the String value to the target type.

707.4.2.2 Conversion to String

Conversion of scalars to String is done by calling toString() on the object to be converted. In the case of a primitive type, the object is boxed first.

A null object results in a null String value.

Exceptions:

- java.util.Calendar and java.util.Date are converted to String as described in *Date and Calendar* on page 1460.
- Map.Entry is converter to String according to the rules in *Map.Entry* on page 1460.

707.4.2.3 Conversion from String

Conversion from String is done by attempting to invoke the following methods, in order:

1. public static valueOf(String s)
2. public constructor taking a single String argument.

Some scalars have special rules for converting from String values. See below.

Table 707.2 Special cases converting to scalars from String

Target	**Method**
char / Character	v.length() > 0 ? v.charAt(0) : 0
java.time.Duration	Duration.parse(v)
java.time.Instant	Instant.parse(v)
java.time.LocalDate	LocalDate.parse(v)
java.time.LocalDateTime	LocalDateTime.parse(v)
java.time.LocalTime	LocalTime.parse(v)
java.time.MonthDay	MonthDay.parse(v)
java.time.OffsetTime	OffsetTime.parse(v)
java.time.OffsetDateTime	OffsetDateTime.parse(v)
java.time.Year	Year.parse(v)
java.time.YearMonth	YearMonth.parse(v)
java.time.ZonedDateTime	ZonedDateTime.parse(v)
java.util.Calendar	See *Date and Calendar* on page 1460.
java.util.Date	See *Date and Calendar* on page 1460.
java.util.UUID	UUID.fromString(v)
java.util.regex.Pattern	Pattern.compile(v)

Note to implementors: some of the classes mentioned in table Table 707.2 are introduced in Java 8. However, a converter implementation does not need to depend on Java 8 in order to function. An implementation of the converter specification could determine its Java runtime dynamically and handle classes in this table depending on availability.

707.4.2.4 Date and Calendar

A `java.util.Date` instance is converted to a `long` value by calling `Date.getTime()`. Converting a `long` into a `java.util.Date` is done by calling `new Date(long)`.

Converting a Date to a String will produce a ISO-8601 UTC date/time string in the following format: `2011-12-03T10:15:30Z`. In Java 8 this can be done by calling `Date.toInstant().toString()`. Converting a String to a Date is done by parsing this ISO-8601 format back into a Date. In Java 8 this function is performed by calling `Date.from(Instant.parse(v))`.

Conversions from Calendar objects are done by converting the Calendar to a Date via `getTime()` first, and then converting the resulting Date to the target type. Conversions to a Calendar object are done by converting the source to a Date object with the desired time (always in UTC) and then setting the time in the Calendar object via `setTime()`.

707.4.2.5 Enums

Conversions to Enum types are supported as follows.

Table 707.3 *Converting to Enum types*

Source	**Method**
Number	*EnumType*.`values()[v.intValue()]`
`String`	*EnumType*.`valueOf(v)`. If this does not produce a result a case-insensitive lookup is done for a matching enum value.

Primitives are boxed before conversion is done. Other source types are converted to String before converting to Enum.

707.4.2.6 Map.Entry

Conversion of `Map.Entry<K,V>` to a target scalar type is done by evaluating the compatibility of the target type with both the key and the value in the entry and then using the best match. This is done in the following order:

1. If one of the key or value is the same as the target type, then this is used. If both are the same, the key is used.
2. If one of the key or value type is assignable to the target type, then this is used. If both are assignable the key is used.
3. If one of the key or value is of type `String`, this is used and converted to the target type. If both are of type `String` the key is used.
4. If none of the above matches the key is converted into a `String` and this value is then converted to the target type.

Note that when applying these rules care must be taken with `null` keys and values. A `null` is not an `Object` and therefore has no type. As a result `null` is never the same type as, or type assignable to, a target type.

Conversion to `Map.Entry` from a scalar is not supported.

707.4.3 Arrays and Collections

This section describes conversions from, to and between Arrays and Collections. This includes Lists, Sets, Queues and Double-ended Queues (*Deques*).

707.4.3.1 Converting from a scalar

Scalars are converted into a Collection or Array by creating an instance of the target type suitable for holding a single element. The scalar source object will be converted to target element type if necessary and then set as the element.

A null value will result in an empty Collection or Array.

Exceptions:

- Converting a String to a char[] or Character[] will result in an array with characters representing the characters in the String.

707.4.3.2 Converting to a scalar

If a Collection or array needs to be converted to a scalar, the first element is taken and converted into the target type. Example:

```
Converter converter = Converters.standardConverter();
String s = converter.convert(new int[] {1,2}).to(String.class)); // s="1"
```

If the collection or array has no elements, the null value is used to convert into the target type.

Note: deviations from this mechanism can be achieved by using a ConverterBuilder. For example:

```
// Use an ConverterBuilder to create a customized converter
ConverterBuilder cb = converter.newConverterBuilder();
cb.rule(new Rule<int[], String>(v -> Arrays.stream(v).
    mapToObj(Integer::toString).collect(Collectors.joining(","))) {});
cb.rule(new Rule<String, int[]>(v -> Arrays.stream(v.split(",")).
    mapToInt(Integer::parseInt).toArray()) {});
Converter c = cb.build();

String s2 = c.convert(new int[] {1,2}).to(String.class)); // s2="1,2"
int[] sa = c.convert("1,2").to(String[].class); // sa={1,2}
```

Exceptions:

- Converting a char[] or Character[] into a String results in a String where each character represents the elements of the character array.

707.4.3.3 Converting to an Array or Collection

When converting to an Array or Collection a separate instance is returned that can be owned by the caller. By default the result is created eagerly and populated with the converted content.

When converting to a java.util.Collection, java.util.List or java.util.Set the converter can produce a live view over the backing object that changes when the backing object changes. The live view can be enabled by specifying the view() modifier.

In all cases the object returned is a separate instance that can be owned by the client. Once the client modifies the returned object a live view will stop reflecting changes to the backing object.

Table 707.4 Collection / Array target creation

Target	Method
Collection interface	A mutable implementation is created. For example, if the target type is java.util.Queue then the converter can create a java.util.LinkedList. When converting to a subinterface of java.util.Set the converter must choose a set implementation that preserves iteration order.

Target	Method
Collection concrete type	A new instance is created by calling Class.newInstance() on the provided type. For example if the target type is Array-Deque then the converter creates a target object by calling ArrayDeque.class.newInstance(). The converter may choose to use a call a well-known constructor to optimize the creation of the collection.
Collection, List or Set with view() modifier	A live view over the backing object is created, changes to the backing object will be reflected, unless the view object is modified by the client.
T[]	A new array is created via Array.newInstance(Class<T> cls, int x) where x is the required size of the target collection.

Before inserting values into the resulting collection/array they are converted to the desired target type. In the case of arrays this is the type of the array. When inserting into a Collection generic type information about the target type can be made available by using the to(TypeReference) or to(Type) methods. If no type information is available, source elements are inserted into the target object as-is without further treatment.

For example, to convert an array of Strings into a list of Integers:

```
List<Integer> result =
  converter.convert(Arrays.asList("1","2","3")).
    to(new TypeReference<List<Integer>>() {});
```

The following example converts an array of ints into a set of Doubles. Note that the resulting set must preserve the same iteration order as the original array:

```
Set<Double> result =
  converter.convert(new int[] {2,3,2,1}).
    to(new TypeReference<Set<Double>>() {})
// result is 2.0, 3.0, 1.0
```

Values are inserted in the target Collection/array as follows:

- If the source object is null, an empty collection/array is produced.
- If the source is a Collection or Array, then each of its elements is converted into desired target type, if known, before inserting. Elements are inserted into the target collection in their normal iteration order.
- If the source is a Map-like structure (as described in *Maps, Interfaces, Java Beans, DTOs and Annotations* on page 1462) then Map.Entry elements are obtained from it by converting the source to a Map (if needed) and then calling Map.entrySet(). Each Map.Entry element is then converted into the target type as described in *Map.Entry* on page 1460 before inserting in the target.

707.4.3.4 Converting to maps

Conversion to a map-like structure from an Array or Collection is not supported by the Standard Converter.

707.4.4 Maps, Interfaces, Java Beans, DTOs and Annotations

Entities that can hold multiple key-value pairs are all treated in a similar way. These entities include Maps, Dictionaries, Interfaces, Java Beans, Annotations and OSGi DTOs. We call these *map-like* types. Additionally objects that provide a map view via getProperties() are supported.

When converting between map-like types, a Map can be used as intermediary. When converting to other, non *map-like*, structures the map is converted into an iteration order preserving collection of Map.Entry values which in turn is converted into the target type.

707.4.4.1 Converting from a scalar

Conversions from a scalar to a map-like type are not supported by the standard converter.

707.4.4.2 Converting to a scalar

Conversions of a map-like structure to a scalar are done by iterating through the entries of the map and taking the first Map.Entry instance. Then this instance is converted into the target scalar type as described in *Map.Entry* on page 1460.

An empty map results in a null scalar value.

707.4.4.3 Converting to an Array or Collection

A map-like structure is converted to an Array or Collection target type by creating an ordered collection of Map.Entry objects. Then this collection is converted to the target type as described in *Arrays and Collections* on page 1460 and *Map.Entry* on page 1460.

Note that due to the rules for converting a Map.Entry into a scalar it is possible that the target Array or Collection may contain elements that were keys from the map, values from the map, or a mixture of the two. To be certain that only keys or values from the map will be used then it is recommended to first convert the map-like structure to a map, and then to convert the keySet or values from that Map into the required target type.

For example:

```
Map<Integer,String> map = Map.of(1, "hi",
      2, null,
      3, "ho");

Converter c = Converters.standardConverter();

List<String> result;

// This result will be ["hi", "2", "ho"]
result = c.convert(map).to(new TypeReference<List<String>>(){})

// This result will be ["1", "2", "3"]
result = c.convert(map.keySet()).to(new TypeReference<List<String>>(){})

// This result will be ["hi", null, "ho"]
result = c.convert(map.values()).to(new TypeReference<List<String>>(){})
```

707.4.4.4 Converting to a map-like structure

Conversions from one map-like structure to another map-like structure are supported. For example, conversions between a map and an annotation, between a DTO and a Java Bean or between one interface and another interface are all supported.

707.4.4.4.1 Key Mapping

When converting to or from a Java type, the key is derived from the method or field name. Certain common property name characters, such as full stop ('.' \u002E) and hyphen-minus ('-' \u002D) are not valid in Java identifiers. So the name of a method must be converted to its corresponding key name as follows:

- A single dollar sign ('$' \u0024) is removed unless it is followed by:
  - A low line ('_' \u005F) and a dollar sign in which case the three consecutive characters ("$_$") are converted to a single hyphen-minus ('-' \u002D).
  - Another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.

- A single low line ('_' \u005F) is converted into a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.
- If the type that declares the method also declares a public static final PREFIX_ field whose value is a compile-time constant String, then the key name is prefixed with the value of the PREFIX_ field. PREFIX_ fields in super-classes or super-interfaces are ignored.

Table 707.5 contains some name mapping examples.

Table 707.5 *Component Property Name Mapping Examples*

Component Property Type Method Name	Component Property Name
myProperty143	myProperty143
$new	new
my$$prop	my$prop
dot_prop	dot.prop
_secret	.secret
another__prop	another_prop
three___prop	three_.prop
four_$__prop	four._prop
five_$_prop	five..prop
six$_$prop	six-prop
seven$$_$prop	seven$.prop

Below is an example of using the PREFIX_ constant in an annotation. The example receives an untyped Dictionary in the updated() callback with configuration information. Each key in the dictionary is prefixed with the PREFIX_. The annotation can be used to read the configuration using typed methods with short names.

```
public @interface MyAnnotation {
  static final String PREFIX_ = "com.acme.config.";

  long timeout() default 1000L;
  String tempdir() default "/tmp";
  int retries() default 10;
}

public void updated(Dictionary dict) {
  // dict contains:
  // "com.acme.config.timeout" = "500"
  // "com.acme.config.tempdir" = "/temp"

  MyAnnotation cfg = converter.convert(dict).to(MyAnnotation.class);

  long configuredTimeout = cfg.timeout(); // 500
  int configuredRetries = cfg.retries(); // 10

  // ...
}
```

However, if the type is a *single-element annotation*, see 9.7.3 in [1] *The Java Language Specification, Java SE 8 Edition*, then the key name for the value method is derived from the name of the component property type rather than the name of the method. In this case, the simple name of the component property type, that is, the name of the class without any package name or outer class name, if the component property type is an inner class, must be converted to the value method's property name as follows:

- When a lower case character is followed by an upper case character, a full stop ('.' \u002E) is inserted between them.
- Each uppercase character is converted to lower case.
- All other characters are unchanged.
- If the annotation type declares a PREFIX_ field whose value is a compile-time constant String, then the id is prefixed with the value of the PREFIX_ field.

Table 707.6 contains some mapping examples for the value method.

Table 707.6 *Single-Element Annotation Mapping Examples for value Method*

Type Name	value **Method Component Property Name**
ServiceRanking	service.ranking
Some_Name	some_name
OSGiProperty	osgi.property

707.4.4.4.2 Converting to a Map

When converting to a Map a separate instance is returned that can be owned by the caller. By default the result is created eagerly and populated with converted content.

When converting to a java.util.Map the converter can produce a live view over the backing object that changes when the backing object changes. The live view can be enabled by specifying the view() modifier.

In all cases the object returned is a separate instance that can be owned by the client. When the client modifies the returned object a live view will stop reflecting changes to the backing object.

Table 707.7 *Map target creation*

Target	**Method**
Map interface	A mutable implementation is created. For example, if the target type is ConcurrentNavigableMap then the implementation can create a ConcurrentSkipListMap.
Map concrete type	A new instance is created by calling Class.newInstance() on the provided type. For example if the target type is HashMap then the converter creates a target object by calling HashMap.class.newInstance(). The converter may choose to use a call a well-known constructor to optimize the creation of the map.
java.util.Map with view() modifier	A map view over the backing object is created, changes to the backing object will be reflected in the map, unless the map is modified by the client.

When converting from a map-like object to a Map or sub-type, each key-value pair in the source map is converted to desired types of the target map using the generic information if available. Map type information for the target type can be made available by using the to(TypeReference) or to(Type) methods. If no type information is available, key-value pairs are used in the map as-is.

707.4.4.4.3 Dictionary

Converting between a map and a Dictionary is done by iterating over the source and inserting the key value pairs in the target, converting them to the requested target type, if known. As with other generic types, target type information for Dictionaries can be provided via a TypeReference.

707.4.4.4.4 Interface

Converting a map-like structure into an interface can be a useful way to give a map of untyped data a typed API. The converter synthesizes an interface instance to represent the conversion.

Note that converting to annotations provides similar functionality with the added benefit of being able to specify default values in the annotation code.

707.4.4.4.4.1 Converting to an Interface

When converting into an interface the converter will create a dynamic proxy to implement the interface. The name of the method returning the value should match the key of the map entry, taking into account the mapping rules specified in *Key Mapping* on page 1463. The key of the map may need to be converted into a String first.

Conversion is done *on demand*: only when the method on the interface is actually invoked. This avoids conversion errors on methods for which the information is missing or cannot be converted, but which the caller does not require.

Note that the converter will not copy the source map when converting to an interface allowing changes to the source map to be reflected live to the proxy. The proxy cannot cache the conversions.

Interfaces can provide methods for default values by providing a single-argument method override in addition to the no-parameter method matching the key name. If the type of the default does not match the target type it is converted first. For example:

```
interface Config {
  int my_value(); // no default
  int my_value(int defVal);
  int my_value(String defVal); // String value is automatically converted to int
  boolean my_other_value();
}

// Usage
Map<String, Object> myMap = new HashMap<>(); // an example map
myMap.put("my.other.value", "true");
Config cfg = converter.convert(myMap).to(Config.class);
int val = cfg.my_value(17); // if not set then use 17
boolean val2 = cfg.my_other_value(); // val2=true
```

Default values are used when the key is not present in the map for the method. If a key is present with a null value, then null is taken as the value and converted to the target type.

If no default is specified and a requested value is not present in the map, a ConversionException is thrown.

707.4.4.4.4.2 Converting from an Interface

An interface can also be the source of a conversion to another map-like type. The name of each method without parameters is taken as key, taking into account the *Key Mapping* on page 1463. The method is invoked using reflection to produce the associated value. Default methods must also be handled by the converter.

Whether a conversion source object is an interface is determined dynamically. When an object implements multiple interfaces by default the first interface from these that has no-parameter methods is taken as the source type. To select a different interface use the sourceAs(Class) modifier:

```
Map m = converter.convert(myMultiInterface).
      sourceAs(MyInterfaceB.class).to(Map.class);
```

If the source object also has a getProperties() method as described in *Types with getProperties()* on page 1468, this getProperties() method is used to obtain the map view by default. This behavior can be overridden by using the sourceAs(Class) modifier.

707.4.4.4.5 Annotation

Conversion to and from annotations behaves similar to interface conversion with the added capability of specifying a default in the annotation definition.

When converting to an annotation type, the converter will return an instance of the requested annotation class. As with interfaces, values are only obtained from the conversion source when the annotation method is actually called. If the requested value is not available, the default as specified in the annotation class is used. If no default is specified a ConversionException is thrown.

Similar to interfaces, conversions to and from annotations also follow the *Key Mapping* on page 1463 for annotation element names. Below a few examples of conversions to an annotation:

```
@interface MyAnnotation {
  String[] args() default {"arg1", "arg2"};
}

// Will set sa={"args1", "arg2"}
String[] sa = converter.convert(new HashMap()).to(MyAnnotation.class).args();

// Will set a={"x", "y", "z"}
Map m = Collections.singletonMap("args", new String [] {"x", "y", "z"});
String[] a = converter.convert(m).to(MyAnnotation.class).args();

// Will set a1={}
Map m1 = Collections.singletonMap("args", null)
String[] a1 = converter.convert(m1).to(MyAnnotation.class).args();

// Will set a2={""}
Map m2 = Collections.singletonMap("args", "")
String[] a2 = converter.convert(m2).to(MyAnnotation.class).args();

// Will set a3={","}
Map m3 = Collections.singletonMap("args", ",")
String[] a3 = converter.convert(m3).to(MyAnnotation.class).args();
```

707.4.4.4.5.1 Marker annotations

If an annotation is a *marker annotation*, see 9.7.2 in [1] *The Java Language Specification, Java SE 8 Edition*, then the property name is derived from the name of the annotation, as described for single-element annotations in *Key Mapping* on page 1463, and the value of the property is Boolean.TRUE.

When converting to a marker annotation the converter checks that the source has key and value that are consistent with the marker annotation. If they are not, for example if the value is not present or does not convert to Boolean.TRUE, then a conversion will result in a Conversion Exception.

707.4.4.4.6 Java Beans

Java Beans are concrete (non-abstract) classes that follow the Java Bean naming convention. They provide public getters and setters to access their properties and have a public no-parameter constructor. When converting from a Java Bean introspection is used to find the read accessors. A read accessor must have no arguments and a non-void return value. The method name must start with get followed by a capitalized property name, for example getSize() provides access to the property size. For boolean/Boolean properties a prefix of is is also permitted. Properties names follow the *Key Mapping* on page 1463.

For the converter to consider an object as a Java Bean the sourceAsBean() or targetAsBean() modifier needs to be invoked, for example:

```
Map m = converter.convert(myBean).sourceAsBean().to(Map.class);
```

When converting to a Java Bean, the bean is constructed eagerly. All available properties are set in the bean using the bean's write accessors, that is, public setters methods with a single argument. All methods of the bean class itself and its super classes are considered. When a property cannot be converted this will cause a ConversionException. If a property is missing in the source, the property will not be set in the bean.

Note: access via indexed bean properties is not supported.

Note: the getClass() method of the java.lang.Object class is not considered an accessor.

707.4.4.4.7 DTOs

DTOs are classes with public non-static fields and no methods other than the ones provided by the java.lang.Object class. OSGi DTOs extend the org.osgi.dto.DTO class, however objects following the DTO rules that do not extend the DTO class are also treated as DTOs by the converter. DTOs may have static fields, or non-public instance fields. These are ignored by the converter.

When converting from a DTO to another map-like structure each public instance field is considered. The field name is taken as the key for the map entry, taking into account *Key Mapping* on page 1463, the field value is taken as the value for the map entry.

When converting to a DTO, the converter attempts to find fields that match the key of each entry in the source map and then converts the value to the field type before assigning it. The key of the map entries may need to be converted into a String first. Keys are mapped according to *Key Mapping* on page 1463.

The DTO is constructed using its no-parameter constructor and each public field is filled with data from the source eagerly. Fields present in the DTO but missing in the source object not be set.

The converter only considers a type to be a DTO type if it declares no methods. However, if a type needs to be treated as a DTO that has methods, the converter can be instructed to do this using the sourceAsDTO() and targetAsDTO() modifiers.

707.4.4.4.8 Types with getProperties()

The converter uses reflection to find a public java.util.Map getProperties() or java.util.Dictionary getProperties() method on the source type to obtain a map view over the source object. This map view is used to convert the source object to a map-like structure.

If the source object both implements an interface and also has a public getProperties() method, the converter uses the getProperties() method to obtain the map view. This getProperties() may or may not be part of an implemented interface.

Note: this mechanism can only be used to convert *to* another type. The reverse is not supported

707.4.4.4.9 Specifying target types

The converter always produces an instance of the target type as specified with the to(Class), to(TypeReference) or to(Type) method. In some cases the converter needs to be instructed how to treat this target object. For example the desired target type might extend a DTO class adding some methods and behavior to the DTO. As this target class now has methods, the converter will not recognize it as a DTO. The targetAs(Class), targetAsBean() and targetAsDTO() methods can be used here to instruct the converter to treat the target object as certain type of object to guide the conversion.

For example:

```
MyExtendedDTO med = converter.convert(someMap).
    targetAsDTO().to(MyExtendedDTO.class)
```

In this example the converter will return a MyExtendedDTO instance but it will treat is as a MyDTO type.

707.5 Repeated or Deferred Conversions

In certain situations the same conversion needs to be performed multiple times, on different source objects. Or maybe the conversion needs to be performed asynchronously as part of a async stream processing pipeline. For such cases the Converter can produce a Function, which will perform the conversion once applied. The function can be invoked multiple times with different source objects. The Converter can produce this function through the function() method, which provides an API similar to the convert(Object) method, with the difference that instead of returning the conversion, once to() is called, a Function that can perform the conversion on apply(T) is returned.

The following example sets up a Function that can perform conversions to Integer objects. A default value of 999 is specified for the conversion:

```
Converter c = Converters.standardConverter();

// Obtain a function for the conversion
Function<Object, Integer> cf = c.function().defaultValue(999).to(Integer.class);

// Use the converter multiple times:
Integer i1 = cf.apply("123"); // i1 = 123
Integer i2 = cf.apply("");    // i2 = 999
```

The Function returned by the converter is thread safe and can be used concurrently or asynchronously in other threads.

707.6 Customizing converters

The Standard Converter applies the conversion rules described in this specification. While this is useful for many applications, in some cases deviations from the specified rules may be necessary. This can be done by creating a customized converter. Customized converters are created based on an existing converter with additional rules specified that override the existing converter's behavior. A customized converter is created through a ConverterBuilder. Customized converters implement the converter interface and as such can be used to create further customized converters. Converters are immutable, once created they cannot be modified, so they can be freely shared without the risk of modification to the converter's behavior.

For example converting a Date to a String may require a specific format. The default Date to String conversion produces a String in the format yyyy-MM-ddTHH:mm:ss.SSSZ. If we want to produce a String in the format yyMMddHHmmssZ instead a custom converter can be applied:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyMMddHHmmssZ") {
  @Override
  public synchronized StringBuffer format(Date date, StringBuffer toAppendTo,
                                          FieldPosition pos) {
    // Make the method synchronized to support multi threaded access
    return super.format(date, toAppendTo, pos);
  }
};
ConverterBuilder cb = Converters.newConverterBuilder();
cb.rule(new TypeRule<>(Date.class, String.class, sdf::format));
Converter c = cb.build();

String s = c.convert(new Date()).to(String.class);
// s = "160923102853+0100" or similar
```

Custom conversions are also applied to embedded conversions that are part of a map or other enclosing object:

```
class MyBean {
  //... fields ommitted
  boolean getEnabled() { /* ... */ }
  void setEnabled(boolean e)  { /* ... */ }
  Date getStartDate() { /* ... */ }
  void setStartDate(Date d) { /* ... */ }
}

MyBean mb = new MyBean();
mb.setStartDate(new Date());
mb.setEnabled(true);

Map<String, String> m = c.convert(mb).sourceAsBean().
    to(new TypeReference<Map<String, String>>(){});
String en = m.get("enabled");   // en = "true"
String sd = m.get("startDate"); // sd = "160923102853+0100" or similar
```

A converter rule can return CANNOT_HANDLE to indicate that it cannot handle the conversion, in which case next applicable rule is handed the conversion. If none of the registered rules for the current converter can handle the conversion, the parent converter object is asked to convert the value. Since custom converters can be the basis for further custom converters, a chain of custom converters can be created where a custom converter rule can either decide to handle the conversion, or it can delegate back to the next converter in the chain by returning CANNOT_HANDLE if it wishes to do so.

707.6.1 Catch-all rules

It is also possible to register converter rules which are invoked for every conversion with the rule(ConverterFunction) method. When multiple rules are registered, they are evaluated in the order of registration, until a rule indicates that it can handle a conversion. A rule can indicate that it cannot handle the conversion by returning the CANNOT_HANDLE constant. Rules targeting specific types are evaluated before catch-all rules.

707.7 Conversion failures

Not all conversions can be performed by the standard converter. It cannot convert text such as 'lorem ipsum' into a long value. Or the number *pi* into a map. When a conversion fails, the converter will throw a ConversionException.

If meaningful conversions exist between types not supported by the standard converter, a customized converter can be used, see *Customizing converters* on page 1469.

Some applications require different behavior for error scenarios. For example they can use an empty value such as 0 or "" instead of the exception, or they might require a different exception to be thrown. For these scenarios a custom error handler can be registered. The error handler is only invoked in cases where otherwise a ConversionException would be thrown. The error handler can return a different value instead or throw another exception.

An error handler is registered by creating a custom converter and providing it with an error handler via the errorHandler(ConverterFunction) method. When multiple error handlers are registered for a given converter they are invoked in the order in which they were registered until an error handler either throws an exception or returns a value other than CANNOT_HANDLE.

707.8 Security

An implementation of this specification will require the use of Java Reflection APIs. Therefore it should have the appropriate permissions to perform these operations when running under the Java Security model.

707.9 org.osgi.util.converter

Converter Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.util.converter; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.util.converter; version="[1.0,1.1)"
```

707.9.1 Summary

- ConversionException - This Runtime Exception is thrown when an object is requested to be converted but the conversion cannot be done.
- Converter - The Converter service is used to start a conversion.
- ConverterBuilder - A builder to create a new converter with modified behavior based on an existing converter.
- ConverterFunction - An functional interface with a convert method that is passed the original object and the target type to perform a custom conversion.
- Converters - Factory class to obtain the standard converter or a new converter builder.
- Converting - This interface is used to specify the target that an object should be converted to.
- Functioning - This interface is used to specify the target function to perform conversions.
- Rule - A rule implementation that works by capturing the type arguments via subclassing.
- Specifying - This is the base interface for the Converting and Functioning interfaces and defines the common modifiers that can be applied to these.
- TargetRule - Interface for custom conversion rules.
- TypeReference - An object does not carry any runtime information about its generic type.
- TypeRule - Rule implementation that works by passing in type arguments rather than subclassing.

707.9.2 public class ConversionException extends RuntimeException

This Runtime Exception is thrown when an object is requested to be converted but the conversion cannot be done. For example when the String "test" is to be converted into a Long.

707.9.2.1 public ConversionException(String message)

message The message for this exception.

□ Create a Conversion Exception with a message.

707.9.2.2 public ConversionException(String message, Throwable cause)

message The message for this exception.

cause The causing exception.

□ Create a Conversion Exception with a message and a nested cause.

707.9.3 public interface Converter

The Converter service is used to start a conversion. The service is obtained from the service registry. The conversion is then completed via the Converting interface that has methods to specify the target type.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

707.9.3.1 public Converting convert(Object obj)

obj The object that should be converted.

□ Start a conversion for the given object.

Returns A Converting object to complete the conversion.

707.9.3.2 public Functioning function()

□ Start defining a function that can perform given conversions.

Returns A Functioning object to complete the definition.

707.9.3.3 public ConverterBuilder newConverterBuilder()

□ Obtain a builder to create a modified converter based on this converter. For more details see the ConverterBuilder interface.

Returns A new Converter Builder.

707.9.4 public interface ConverterBuilder

A builder to create a new converter with modified behavior based on an existing converter. The modified behavior is specified by providing rules and/or conversion functions. If multiple rules match they will be visited in sequence of registration. If a rule's function returns null the next rule found will be visited. If none of the rules can handle the conversion, the original converter will be used to perform the conversion.

Provider Type Consumers of this API must not implement this type

707.9.4.1 public Converter build()

□ Build the specified converter. Each time this method is called a new custom converter is produced based on the rules registered with the builder.

Returns A new converter with the rules provided to the builder.

707.9.4.2 public ConverterBuilder errorHandler(ConverterFunction func)

func The function to be used to handle errors.

□ Register a custom error handler. The custom error handler will be called when the conversion would otherwise throw an exception. The error handler can either throw a different exception or return a value to be used for the failed conversion.

Returns This converter builder for further building.

707.9.4.3 public ConverterBuilder rule(Type type, ConverterFunction func)

type The type that this rule will produce.

func The function that will handle the conversion.

□ Register a conversion rule for this converter. Note that only the target type is specified, so the rule will be visited for every conversion to the target type.

Returns This converter builder for further building.

707.9.4.4 public ConverterBuilder rule(TargetRule rule)

rule A rule implementation.

□ Register a conversion rule for this converter.

Returns This converter builder for further building.

707.9.4.5 public ConverterBuilder rule(ConverterFunction func)

func The function that will handle the conversion.

□ Register a catch-all rule, will be called of no other rule matches.

Returns This converter builder for further building.

707.9.5 public interface ConverterFunction

An functional interface with a convert method that is passed the original object and the target type to perform a custom conversion.

This interface can also be used to register a custom error handler.

707.9.5.1 public static final Object CANNOT_HANDLE

Special object to indicate that a custom converter rule or error handler cannot handle the conversion.

707.9.5.2 public Object apply(Object obj, Type targetType) throws Exception

obj The object to be converted. This object will never be `null` as the convert function will not be invoked for null values.

targetType The target type.

□ Convert the object into the target type.

Returns The conversion result or CANNOT_HANDLE to indicate that the convert function cannot handle this conversion. In this case the next matching rule or parent converter will be given a opportunity to convert.

Throws `Exception`– the operation can throw an exception if the conversion can not be performed due to incompatible types.

707.9.6 public class Converters

Factory class to obtain the standard converter or a new converter builder.

Concurrency Thread-safe

707.9.6.1 public static ConverterBuilder newConverterBuilder()

□ Obtain a converter builder based on the standard converter.

Returns A new converter builder.

707.9.6.2 public static Converter standardConverter()

□ Obtain the standard converter.

Returns The standard converter.

707.9.7 public interface Converting extends Specifying<Converting>

This interface is used to specify the target that an object should be converted to. A Converting instance can be obtained via the Converter.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

707.9.7.1 public T to(Class<T> cls)

Type Parameters <T>

<T> The type to convert to.

cls The class to convert to.

□ Specify the target object type for the conversion as a class object.

Returns The converted object.

707.9.7.2 public T to(Type type)

Type Parameters <T>

<T> The type to convert to.

type A Type object to represent the target type to be converted to.

□ Specify the target object type as a Java Reflection Type object.

Returns The converted object.

707.9.7.3 public T to(TypeReference<T> ref)

Type Parameters <T>

<T> The type to convert to.

ref A type reference to the object being converted to.

□ Specify the target object type as a TypeReference. If the target class carries generics information a TypeReference should be used as this preserves the generic information whereas a Class object has this information erased. Example use:

```
List<String> result = converter.convert(Arrays.asList(1, 2, 3))
  .to(new TypeReference<List<String>>() {
  });
```

Returns The converted object.

707.9.8 public interface Functioning extends Specifying<Functioning>

This interface is used to specify the target function to perform conversions. This function can be used multiple times. A Functioning instance can be obtained via the Converter.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

707.9.8.1 public Function<Object, T> to(Class<T> cls)

Type Parameters <T>

<T> The type to convert to.

cls The class to convert to.

□ Specify the target object type for the conversion as a class object.

Returns A function that can perform the conversion.

707.9.8.2 public Function<Object, T> to(Type type)

Type Parameters <T>

<T> The type to convert to.

type A Type object to represent the target type to be converted to.

□ Specify the target object type as a Java Reflection Type object.

Returns A function that can perform the conversion.

707.9.8.3 public Function<Object, T> to(TypeReference<T> ref)

Type Parameters <T>

<T> The type to convert to.

ref A type reference to the object being converted to.

□ Specify the target object type as a TypeReference. If the target class carries generics information a TypeReference should be used as this preserves the generic information whereas a Class object has this information erased. Example use:

```
List<String> result = converter.function()
  .to(new TypeReference<List<String>>() {
  });
```

Returns A function that can perform the conversion.

707.9.9 public abstract class Rule<F, T> implements TargetRule

<F> The type to convert from.

<T> The type to convert to.

A rule implementation that works by capturing the type arguments via subclassing. The rule supports specifying both *from* and *to* types. Filtering on the *from* by the Rule implementation. Filtering on the *to* is done by the converter customization mechanism.

707.9.9.1 public Rule(Function<F, T> func)

func The conversion function to use.

□ Create an instance with a conversion function.

707.9.9.2 public ConverterFunction getFunction()

□ The function to perform the conversion.

Returns The function.

707.9.9.3 public Type getTargetType()

□ The target type of this rule. The conversion function is invoked for each conversion to the target type.

Returns The target type.

707.9.10 public interface Specifying<T extends Specifying<T>>

<T> Either Converting or Specifying.

This is the base interface for the Converting and Functioning interfaces and defines the common modifiers that can be applied to these.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

707.9.10.1 public T extends Specifying<T> defaultValue(Object defVal)

defVal The default value.

□ The default value to use when the object cannot be converted or in case of conversion from a `null` value.

Returns The current `Converting` object so that additional calls can be chained.

707.9.10.2 public T extends Specifying<T> keysIgnoreCase()

□ When converting between map-like types use case-insensitive mapping of keys.

Returns The current `Converting` object so that additional calls can be chained.

707.9.10.3 public T extends Specifying<T> sourceAs(Class<?> cls)

cls The class to treat the object as.

□ Treat the source object as the specified class. This can be used to disambiguate a type if it implements multiple interfaces or extends multiple classes.

Returns The current `Converting` object so that additional calls can be chained.

707.9.10.4 public T extends Specifying<T> sourceAsBean()

□ Treat the source object as a JavaBean. By default objects will not be treated as JavaBeans, this has to be specified using this method.

Returns The current `Converting` object so that additional calls can be chained.

707.9.10.5 public T extends Specifying<T> sourceAsDTO()

□ Treat the source object as a DTO even if the source object has methods or is otherwise not recognized as a DTO.

Returns The current `Converting` object so that additional calls can be chained.

707.9.10.6 public T extends Specifying<T> targetAs(Class<?> cls)

cls The class to treat the object as.

□ Treat the target object as the specified class. This can be used to disambiguate a type if it implements multiple interfaces or extends multiple classes.

Returns The current Converting object so that additional calls can be chained.

707.9.10.7 public T extends Specifying<T> targetAsBean()

□ Treat the target object as a JavaBean. By default objects will not be treated as JavaBeans, this has to be specified using this method.

Returns The current Converting object so that additional calls can be chained.

707.9.10.8 public T extends Specifying<T> targetAsDTO()

□ Treat the target object as a DTO even if it has methods or is otherwise not recognized as a DTO.

Returns The current Converting object so that additional calls can be chained.

707.9.10.9 public T extends Specifying<T> view()

□ Return a live view over the backing object that reflects any changes to the original object. This is only possible with conversions to java.util.Map, java.util.Collection, java.util.List and java.util.Set. The live view object will cease to be live as soon as modifications are made to it. Note that conversions to an interface or annotation will always produce a live view that cannot be modified. This modifier has no effect with conversions to other types.

Returns The current Converting object so that additional calls can be chained.

707.9.11 public interface TargetRule

Interface for custom conversion rules.

707.9.11.1 public ConverterFunction getFunction()

□ The function to perform the conversion.

Returns The function.

707.9.11.2 public Type getTargetType()

□ The target type of this rule. The conversion function is invoked for each conversion to the target type.

Returns The target type.

707.9.12 public class TypeReference<T>

<T> The target type for the conversion.

An object does not carry any runtime information about its generic type. However sometimes it is necessary to specify a generic type, that is the purpose of this class. It allows you to specify an generic type by defining a type T, then subclassing it. The subclass will have a reference to the super class that contains this generic information. Through reflection, we pick this reference up and return it with the getType() call.

```
List<String> result = converter.convert(Arrays.asList(1, 2, 3))
  .to(new TypeReference<List<String>>() {
  });
```

Concurrency Immutable

707.9.12.1 protected TypeReference()

□ A TypeReference cannot be directly instantiated. To use it, it has to be extended, typically as an anonymous inner class.

707.9.12.2 public Type getType()

□ Return the actual type of this Type Reference

Returns the type of this reference.

707.9.13 public class TypeRule<F, T> implements TargetRule

<F> The type to convert from.

<T> The type to convert to.

Rule implementation that works by passing in type arguments rather than subclassing. The rule supports specifying both *from* and *to* types. Filtering on the *from* by the `Rule` implementation. Filtering on the *to* is done by the converter customization mechanism.

707.9.13.1 public TypeRule(Type from, Type to, Function<F, T> func)

from The type to convert from.

to The type to convert to.

func The conversion function to use.

□ Create an instance based on source, target types and a conversion function.

707.9.13.2 public ConverterFunction getFunction()

□ The function to perform the conversion.

Returns The function.

707.9.13.3 public Type getTargetType()

□ The target type of this rule. The conversion function is invoked for each conversion to the target type.

Returns The target type.

707.10 References

[1] *The Java Language Specification, Java SE 8 Edition*
https://docs.oracle.com/javase/specs/jls/se8/html/index.html

End Of Document