Automatically managing service dependencies in OSGi

Marcel Offermans
luminis®
marcel.offermans@luminis.nl

January 7, 2005

Contents

1 Introduction **1**

2 Goals **3**

3 Design **3**

4 Usage **5**
4.1 Registering a service 5
4.2 Specifying service dependencies 6
4.3 Implementing the service 6
4.4 Listening to service dependencies 7
4.5 Listening to the service state 8
4.6 Interacting with OSGi 8
4.7 Dynamic dependencies 9

5 Implementation **9**

6 Conclusion **9**

A White board pattern **10**

B Null object pattern **10**

C Code example **10**

1 Introduction

In a service oriented architecture, applications consist of several cooperating services. These networks of services are often dynamic

in nature, so managing dependencies is something the developer needs to take into account. In an OSGi framework, services are deployed using bundles and these bundles feature two types of dependencies:

1. Package dependencies. A bundle can export a package which others import. These dependencies, although dynamic, are relatively easy to handle.

2. Service dependencies. Services, encapsulated in deployable components (bundles) can be started and stopped any time. Other components often depend on these services and need to deal with changes in their availability.

When you look at dependency management, there are two aspects you need to take into account:

The first is managing software configurations. This means you need to manage the dependencies from a configuration standpoint. What you are managing are bundles, since those are the units of deployment. What you need to manage are the package and service dependencies between bundles. Package dependencies are always visible by examining the bundle manifest and when a bundle is installed the framework will try to resolve such dependencies before that bundle can even be started. Service dependencies are only optionally described in the manifest [OSGi-R3][p. 76 - Importing and Exporting Services] by a list of services a bundle might export as well as a list it might use (import). The words 'optionally' and 'might' already indicate that these aren't things we can depend on. The framework doesn't have to perform any checks on these attributes.

The second is managing service dependencies at runtime. As mentioned before, a service oriented architecture is dynamic by design, so your implementation should be able to deal with this. Bundles can start in any order and any service can go away or be replaced by a different implementation at any point in time. OSGi itself offers basic assistance for tracking services. You can track them yourself by registering as a service listener. A slightly more advanced way is to create a service tracker, which you can subsequently query, or have it notify you on changes. A third alternative is the service binder [SBINDER], which uses XML component descriptors to specify the dependencies.

In real implementations, you are probably going to track multiple services. Using service trackers in such a scenario has the tendency to result in dependency logic that is entangled in the implementation instead of being expressed in a declarative way, as can be seen

in the code example in appendix C. Using a declarative way to specify dependencies has clear advantages when it comes to monitoring and managing them, a task that becomes more and more important in modern, federated, service oriented environments. This article presents a solution that allows you to just define the dependencies and let a dependency manager do all the hard work for you.

2 Goals

This article presents an automated dependency manager for the OSGi framework. It's design was driven by two important goals:

1. Provide a clean separation between a service implementation and the "glue" that binds it to the OSGi framework. The service implementation should not have to contain any OSGi specific code.

2. Minimize the amount of code that needs to be written. The specification and management of dependencies should be automated as much as possible, whilst still providing enough flexibility to customize the system.

3 Design

The proposed solution is a dedicated component for managing dependencies. When using service trackers or listeners, a substantial amount of code needs to be written to track other services and act on changes. From a logical point of view, through a bundle you're managing one or more services. Each service can have zero or more dependencies that can either be required or optional. Required dependencies must be available before the service can even do its work, optional dependencies are used only if they're available.

So as a programmer, you just want to specify these services and dependencies. The design of the dependency manager focusses on this. The class diagram in figure 1 shows the `DependencyActivatorBase`, which is an implementation of `BundleActivator`. You implement the `init()` and `destroy()` methods which are directly related to the `start()` and `stop()` methods of the activator. The `DependencyManager`, which is passed as an argument to these methods, can then be used to define the services and dependencies. It takes care of tracking and activating the service once the dependencies have been resolved.

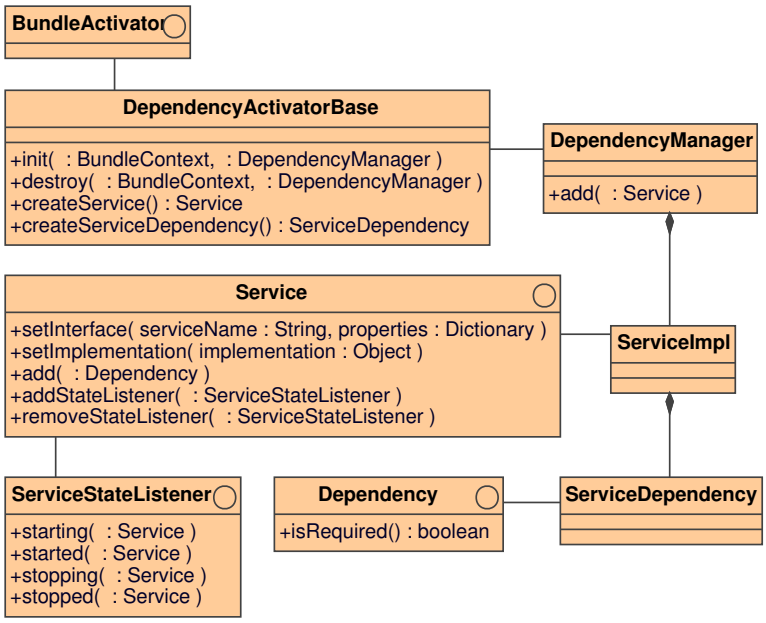

Figure 1: Class diagram.

Effectively, this mechanism extends the state diagram for a bundle, as shown in [OSGi-R3][p. 58 fig 18]. The overall state diagram can now be seen in figure 2. As soon as the bundle is in the active state, it starts tracking required dependencies. When they're resolved, the bundle starts tracking the optional dependencies.

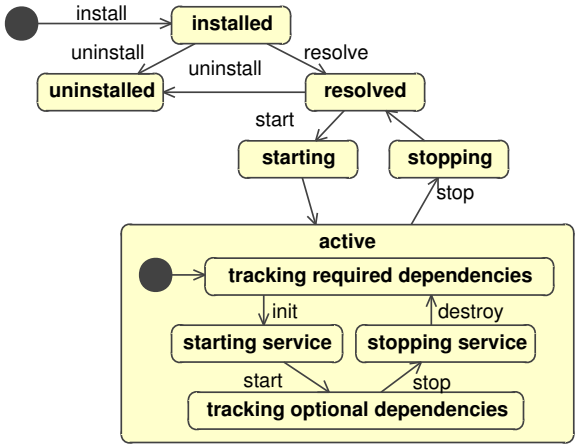

Figure 2: Bundle and service life-cycle.

4 Usage

Let's look at how to use this implementation. When developing an OSGi bundle that uses and possibly registers services, there are two classes in particular we need to implement. The first one is the bundle activator which controls the life-cycle of the bundle. The second one is the actual service implementation.

When using the dependency manager, your bundle activator is a subclass of `DependencyActivatorBase`. The following paragraphs will explain how to specify a service and its dependencies. Subsequently, some more advanced scenarios will be covered that involve listening to dependency and service state changes and interacting with the OSGi framework from within your service implementation.

4.1 Registering a service

Registering a service is done using a custom implementation of the bundle activator, `DependencyActivatorBase`, which requires you to implement two methods: `init` and `destroy`.

Each bundle can specify zero or more services in the `init` method. Services can optionally have an interface and properties with which they're registered into the service registry. Registering a service involves creating a new service and adding it to the dependency manager. You then specify the interface as well as the implementation, as shown in the example.

```
1  public HttpActivator extends DependencyActivatorBase {
2    public void init(BundleContext ctx, DependencyManager manager)
3                                    throws Exception {
4      Properties props = new Properties();
5      props.put("port", "8080");
6      manager.add(createService()
7        .setInterface(WebService.class.getName(), props)
8        .setImplementation(WebServiceImpl.class));
9    }
10   public void destroy(BundleContext ctx, DependencyManager manager)
11                                   throws Exception {
12     // cleaned up automatically
13   }
14 }
```

There are a couple of alternatives that need some clarification. First of all, you might not have a public interface at all, so the `setInterface` call is optional. Like in OSGi itself, you can also specify multiple service interfaces in an array of strings. For the implementation you have two choices. You can either specify the class of the implementation, in which case this class needs to have

a default, no-args constructor and the dependency manager will use lazy instantiation or you can specify an instance you've already created. More on the life-cycle of the service implementation can be found in 4.3.

4.2 Specifying service dependencies

Dependencies can either be required or optional. Required dependencies need to be resolved before the service can even become active. Optional dependencies can appear and disappear while the service is active. The example demonstrates how both are specified in the activator.

```
1     manager.add(createService()
2       .setInterface(WebService.class.getName(), null)
3       .setImplementation(WebServiceImpl.class)
4       .add(createServiceDependency()
5         .setService(ConfigurationAdmin.class, null)
6         .setRequired(true))
7       .add(createServiceDependency()
8         .setService(LogService.class, null)
9         .setRequired(false))
```

4.3 Implementing the service

The service implementation is pretty straightforward. Basically any class that implements the service interface can be used here. Two aspects of the implementation deserve some attention though: the life-cycle model and the service dependencies.

Depending on how the service was specified in the activator it is either instantiated as soon as all required dependencies become available or it was already instantiated when the activator started. The service then goes through a number of states, where each transition includes the invocation of a method on the service implementation. The default names of these methods are `init`, `start`, `stop` and `destroy` but these can be altered. As a developer, all you need to do is specify any of these methods in your service implementation and they will be invoked.

```
1  public class WebServiceImpl implements WebService {
2    public void init() {
3    }
4    public void start() {
5    }
6    public void stop() {
7    }
8    public void destroy() {
9    }
```

```
10 }
```

Upon activation, the service implementation should initialize its internal state in the `init()` method and establish connections with other services in the `start()` method.

The dependency manager will automatically fill in any references to required dependencies that are specified as attributes. The same goes for optional dependencies if they are available. If not, those will be implemented by a null object [NULLOBJ]. In short, this allows you to simply use these interfaces as if they were always available. A good example of this is the `LogService`. If it's available, we want to use it for logging. If not, we want to simply ignore log messages. Normally, you'd need to check a reference to this service for `null` before you can use it. By using a null object, this is not necessary anymore.

```
1  public class WebServiceImpl implements WebService {
2    private ConfigurationAdmin configAdminSvc;
3    private LogService logSvc;
4
5    public void loadSettings() {
6      logSvc.log(LogService.LOG_INFO, "Loading␣settings.");
7      // do stuff here
8    }
9  }
```

4.4 Listening to service dependencies

Optionally, a service can define callbacks for each dependency. These callbacks are invoked whenever a new dependency is discovered or an existing one is disappearing. They allow you to track these dependencies. This can be very useful if you, for example, want to implement the white board pattern.

```
1      manager.add(createService()
2        .setImplementation(DeviceListener.class)
3        .add(createServiceDependency()
4          .setService(Device.class, null)
5          .setCallbacks("added", "removed")
6          .setRequired(false));
```

```
1  public class DeviceListener {
2    public void added(ServiceRegistration reg, Object service) {
3    }
4    public void removed(ServiceRegistration reg, Object service) {
5    }
6  }
```

4.5 Listening to the service state

If you're interested in the state of a service, you can register a service state listener with the service of interest. It will be notified whenever the service is starting, started, stopping and stopped.

```
1     Service s = createService()
2       .setInterface(WebService.class.getName(), null)
3       .setImplementation(WebServiceImpl.class);
4     s.setStateListener(new ServiceStateListener() {
5         public void starting(Service s) {
6         }
7         public void started(Service s) {
8         }
9         public void stopping(Service s) {
10        }
11        public void stopped(Service s) {
12        }
13      });
14    manager.add(s);
```

4.6 Interacting with OSGi

Although interaction of the service implementation with the OSGi framework has nothing to do with dependency managemement, the dependency manager does offer support for communicating with the framework.

Normally, your service implementation does not have to contain any OSGi specific classes or code. As shown in 4.3 even references to other services can be used without dealing with OSGi specifics. Sometimes, this is an advantage, since the service implementation can easily be reused in a different service oriented framework. However, there are times when you do want to access, for example, the BundleContext because you want to interact with the framework.

For these cases, you can specify one or two members to get direct access to:

1. the `BundleContext` that provides you with an interface to the OSGi framework;
2. the `ServiceRegistration` of the service that was registered, provided you have registered a public service interface.

```
1  public class WebServiceImpl implements WebService {
2    private BundleContext ctx;
3    private ServiceRegistration svcReg;
4
5    //
6  }
```

4.7 Dynamic dependencies

Most of the time, the dependency definitions are static. You define them when the bundle starts and they stay the same throughout the life-cycle of the bundle. However, in some cases you might want to add dependencies to a service afterwards. For example, you might want to start tracking a service with specific properties. To do this, simply add the dependency in the same way as you would have done in the `init()` method.

5 Implementation

The dependency manager code is made available under the Common Public Licence 1.0 [CPL]. It can be found on the luminis® open source server [DEPMAN] where a bundle including sourcecode is available for download.

6 Conclusion

When implementing bundles, the dependency manager allows you to specify your dependencies in a declarative way. Not only does this reduce code size, and therefore the potential for errors, it also provides a good foundation for monitoring and management of these dependencies. Furthermore, because of the usage of the null object pattern, service implementations need not check service references for null.

A second advantage is that using the dependency manager makes your bundles more robust with respect to the dynamics of services in the OSGi environment.

References

[OSGi-R3] Open Services Gateway Initiative, *"OSGi Service Platform Release 3"*

[OSGi-WB] Peter Kriens and BJ Hargrave, *"Listener Pattern Considered Harmful – A White Board Paper"*

[SBINDER] Humberto Cervantes and Richard S. Hall, *"Automatic service dependency management in OSGi"* `http://gravity.sourceforge.net/servicebinder/`

[NULLOBJ] *"Null Object"* `http://c2.com/cgi-bin/wiki?NullObject`

[CPL] *"Common Public Licence 1.0"*
`http://www.opensource.org/licenses/cpl.php`

[DEPMAN] *"X-Files Dependency Manager"*
`https://opensource.luminis.net/confluence/display/XF`

A White board pattern

The white board pattern presents a more efficient way to implement listeners. Instead of having listeners track sources and registering themselves with those sources, the white board pattern has the listeners register themselves as services in the OSGi service registry. When a source needs to notify listeners, it simply looks up all listener services in the registry and notifies them.

The pattern is explained in great detail in [OSGi-WB] where both the traditional listener implementation and the white board implementation are compared.

B Null object pattern

A null object pattern [NULLOBJ] provides an object as a surrogate for the lack of an object of a given type. It essentially provides intelligent "do nothing" behavior, hiding the details from its collaborators. The pattern is also known as "Stub" or "Active Nothing".

The motivation for using it is that sometimes a class that requires a collaborator does not need the collaborator to do anything. However, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

C Code example

This appendix shows two bundle activators. The first uses the standard OSGi service trackers to manage dependencies, the second uses the dependency manager. Both activators monitor two required service dependencies `AudioSource` and `AudioEncoder` and one optional dependency `LogService`. They register a service called `AudioBroadcaster` as soon as the required dependencies have been found.

```
1  public class Activator implements BundleActivator {
2    private BundleContext context;
3    private ServiceRegistration registration;
4    private AudioBroadcaster audioBroadcaster;
5    private AudioSource audioSource;
```

```
6     private AudioEncoder audioEncoder;
7     private ServiceTracker audioSourceTracker;
8     private ServiceTracker audioEncoderTracker;
9     private ServiceTracker logTracker;
10
11    public void start(BundleContext context) throws Exception {
12      this.context = context;
13      audioSourceTracker = new ServiceTracker(context,
14          AudioSource.class.getName(), customizer);
15      audioEncoderTracker = new ServiceTracker(context,
16          AudioEncoder.class.getName(), customizer);
17      logTracker = new ServiceTracker(context, LogService.class.getName(),
18          null);
19      logTracker.open();
20      audioSourceTracker.open();
21      audioEncoderTracker.open();
22    }
23
24    public void stop(BundleContext context) throws Exception {
25      audioSourceTracker.close();
26      audioEncoderTracker.close();
27      logTracker.close();
28    }
29
30    private ServiceTrackerCustomizer customizer =
31        new ServiceTrackerCustomizer() {
32      public Object addingService(ServiceReference reference) {
33        Object service = context.getService(reference);
34        setService(reference, service);
35        return service;
36      }
37
38      private void setService(ServiceReference reference, Object service) {
39        // update service references
40        Object objectclass = reference.getProperty(Constants.OBJECTCLASS);
41        if (objectclass instanceof String) {
42          String name = (String) objectclass;
43          setNamedService(service, name);
44        }
45        if (objectclass instanceof String[]) {
46          String[] names = (String[]) objectclass;
47          for (int i = 0; i < names.length; i++) {
48            setNamedService(service, names[i]);
49          }
50        }
51
52        // register service if necessary
53        if ((registration == null) && (audioSource != null)
54            && (audioEncoder != null)) {
55          // instantiate the implementation and pass the services
56          audioBroadcaster = new AudioBroadcasterImpl(audioSource,
57              audioEncoder, logTracker);
58          registration = context.registerService(
59              AudioBroadcaster.class.getName(), audioBroadcaster, null);
60        }
61
62        // unregister service if necessary
63        if (((audioSource == null) || (audioEncoder == null))
64            && (registration != null)) {
65          registration.unregister();
66          registration = null;
67          audioBroadcaster = null;
```

```
68        }
69      }
70
71      private void setNamedService(Object service, String name) {
72        if (AudioEncoder.class.getName().equals(name)) {
73          audioEncoder = (AudioEncoder) service;
74        }
75        else if (AudioSource.class.getName().equals(name)) {
76          audioSource = (AudioSource) service;
77        }
78      }
79
80      public void modifiedService(ServiceReference reference,
81          Object service) {
82      }
83
84      public void removedService(ServiceReference reference,
85          Object service) {
86        setService(reference, null);
87        context.ungetService(reference);
88      }
89    };
90  }
```

```
1   public class Activator extends DependencyActivatorBase {
2    public void init(BundleContext ctx, DependencyManager manager)
3                                                   throws Exception {
4      manager.add(createService()
5        .setInterface(AudioBroadcaster.class.getName(), null)
6        .setImplementation(AudioBroadcasterImpl.class));
7        .add(createServiceDependency()
8          .setService(AudioSource.class, null)
9          .setRequired(true))
10       .add(createServiceDependency()
11         .setService(AudioEncoder.class, null)
12         .setRequired(true))
13       .add(createServiceDependency()
14         .setService(LogService.class, null)
15         .setRequired(false))
16   }
17
18   public void destroy(BundleContext ctx, DependencyManager manager)
19                                                       throws Exception {
20   }
21  }
```