

# **OSGi Alliance Community Event**

## **Enterprise Persistence in OSGi**

**Mike Keith  
Oracle Corporation**



## Who Am I?

---

- Java and Persistence architect @ Oracle
- Have managed to stay on the server side for most of my professional life
- Specialist in areas of distributed systems, transactions and persistence
- Work with numerous expert groups and committees (JCP, OSOA, OASIS, OSGi)
- EclipseLink project committer and mentor



# Agenda

---

- Goals
- Background – EclipseLink and JPA
- Use Cases
- Problems Encountered
- Solutions (adopted or otherwise)
- Summary



## Goals

---

- Make EclipseLink OSGi-enabled
  - JPA implementation working in OSGi
  - Other EclipseLink components as well (JAXB, SDO)
  - Agnostic to particular OSGi implementation (portable)
- Allow OSGi-based applications to use JPA
  - Core application code should not require modification in order to run in OSGi
  - Applications should be able to leverage OSGi model
- Establish patterns and services for general use
  - Work with others to come up with consensus path
  - Feed results back into specification





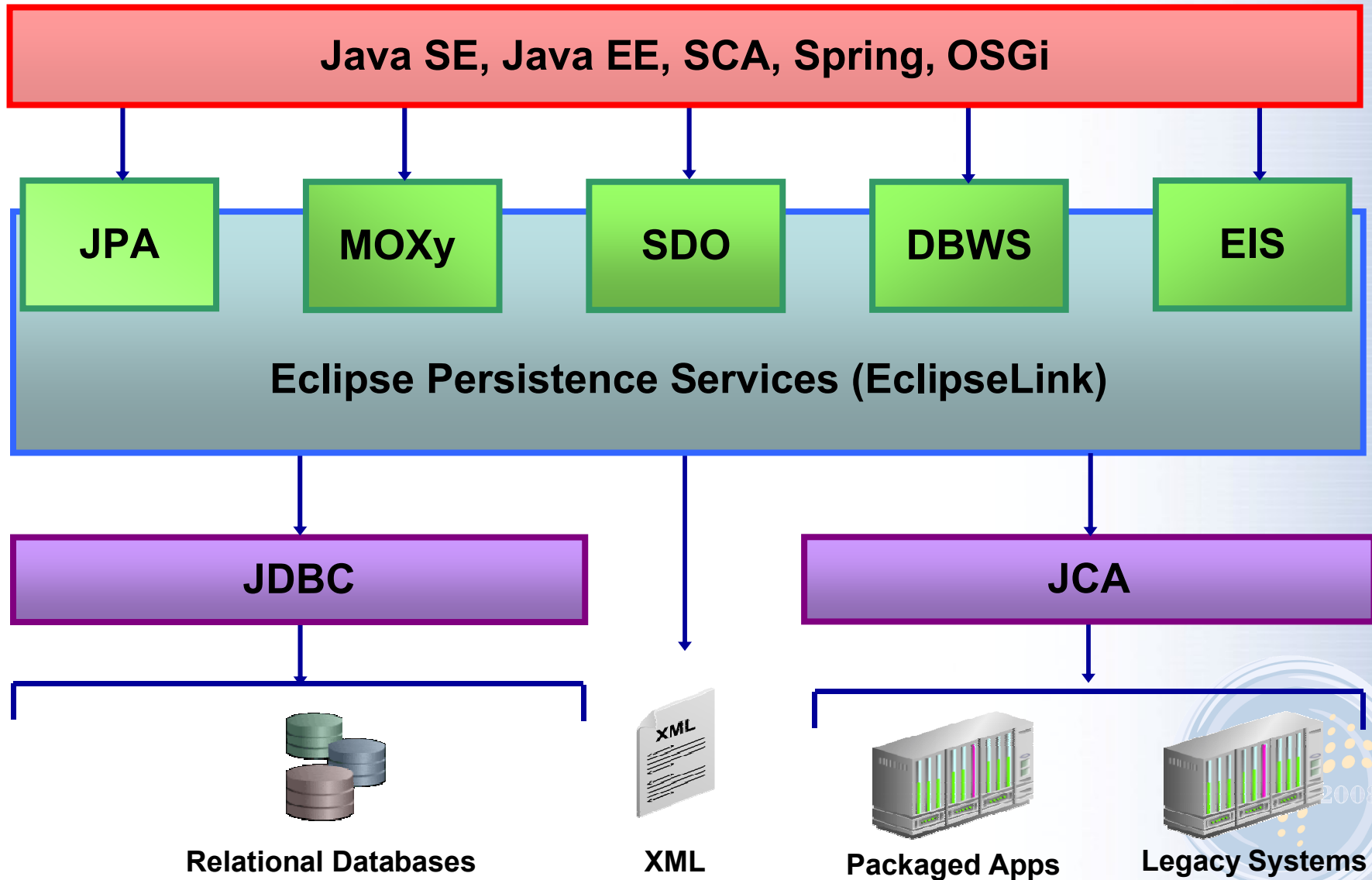
Runtime,  
not tooling!

## EclipseLink Project

---

- Open source project at Eclipse
- Formally called **Eclipse Persistence Services**
- Fully featured open source evolution of TopLink
- Runtime project, not coupled to IDE
- Implements standards across all aspects of persistence:
  - JPA - Relational databases,
  - JAXB, SDO – XML schemas
  - JCA – EIS and other types of non-relational stores
- Also offers XVA (extreme value-add) in all areas
- One-stop shop for persistence needs of any application





## EclipseLink Project

---

- Includes **Eclipse JPA**
  - Fully compliant JPA implementation
  - JPA 2.0 Reference Implementation (RI)
  - Plus a plethora of extra features
    - Cache coordination for clustered apps
    - Deluxe customizable mappings
    - Built-in support for Oracle database features
    - Full Java expression-based query language
    - Complete stored procedure support
    - Many performance tuning enhancements



## JPA On a Slide

---

- Application creates persistent domain objects called “entities” which map to rows in database
- Simple configuration file called persistence.xml
- EntityManager API to control persistence and life cycle of the entities
- Runs in managed or non-managed environments
- Container offers a number of ease of use services
- Underlying persistence provider offers the implementation of the specification



# Pluggable Persistence

---

```
package javax.persistence.spi;  
  
public interface PersistenceProvider {  
  
    public EntityManagerFactory  
        createEntityManagerFactory(  
            String puName, Map map);  
  
    public EntityManagerFactory  
        createContainerEntityManagerFactory (  
            PersistenceUnitInfo info, Map map);  
}
```



# Container Mode

---

JPA can be used in two different modes

1. Container mode:

```
@Stateless
public class MySessionBean {
    @PersistenceContext EntityManager em;

    public Employee getEmployee(int empId) {
        return em.find(Employee.class, empId);
    }
    ...
}
```



# Persistence Info

---

```
package javax.persistence.spi;

public interface PersistenceUnitInfo {

    ...

    public DataSource getJtaDataSource();
    public DataSource getNonJtaDataSource();
    public List<String> getMappingFileNames();
    public List<URL> getJarFileUrls();
    public URL getPersistenceUnitRootUrl();
    public List<String> getManagedClassNames();
    public Properties getProperties();
    public ClassLoader getClassLoader();
    public void addTransformer(ClassTransformer transformer);
    public ClassLoader getNewTempClassLoader();

}
```



# Non-container Mode

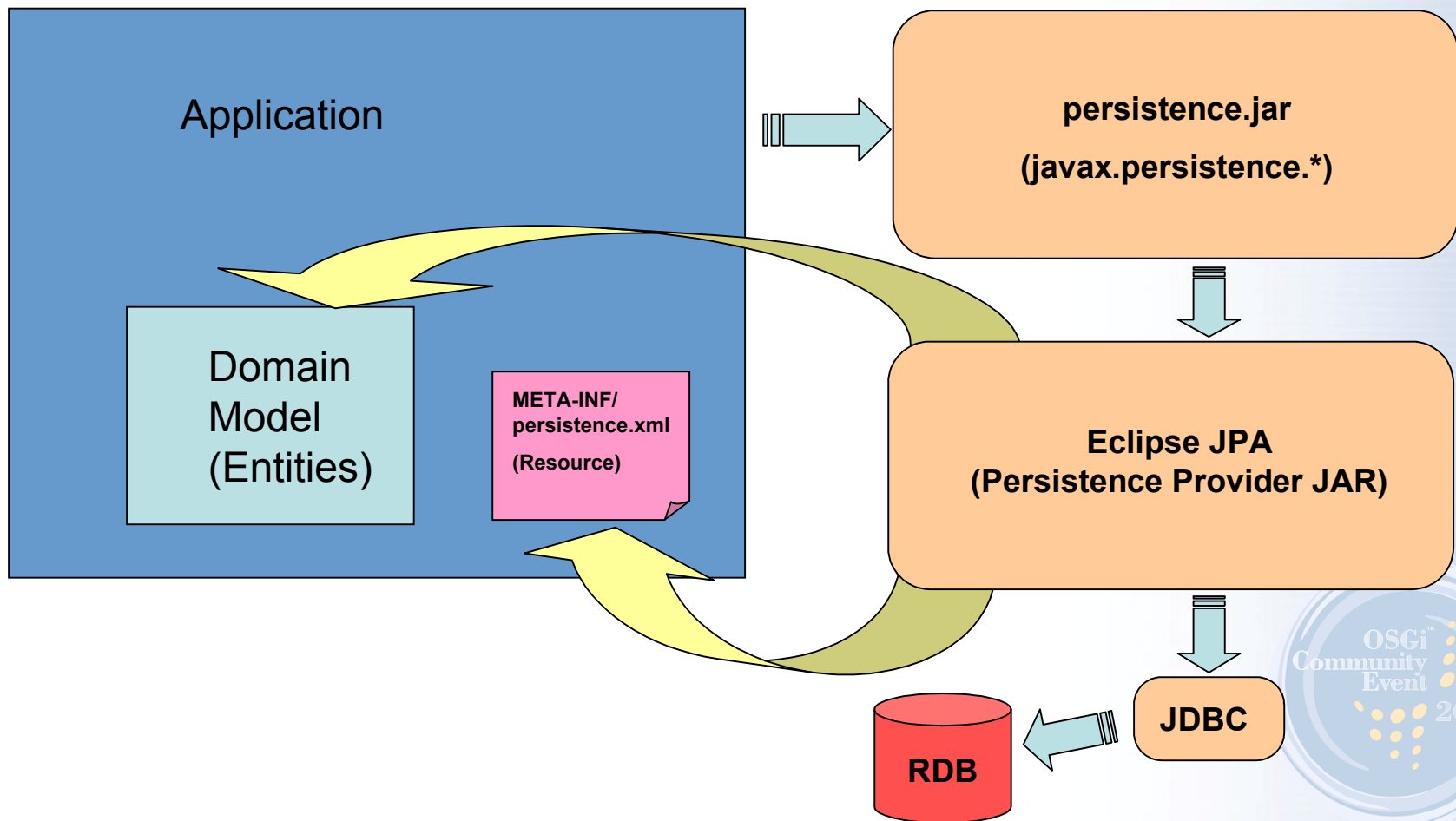
---

## 2. Java SE mode:

```
public class DisplayEmployeeInfo {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("PUnit");
        EntityManager em = emf.createEntityManager();
        Employee emp = em.find(Employee.class,
            new Integer(args[0]));
        System.out.println("EmployeeSummary: " + emp);
        em.close();
        emf.close();
    }
}
```

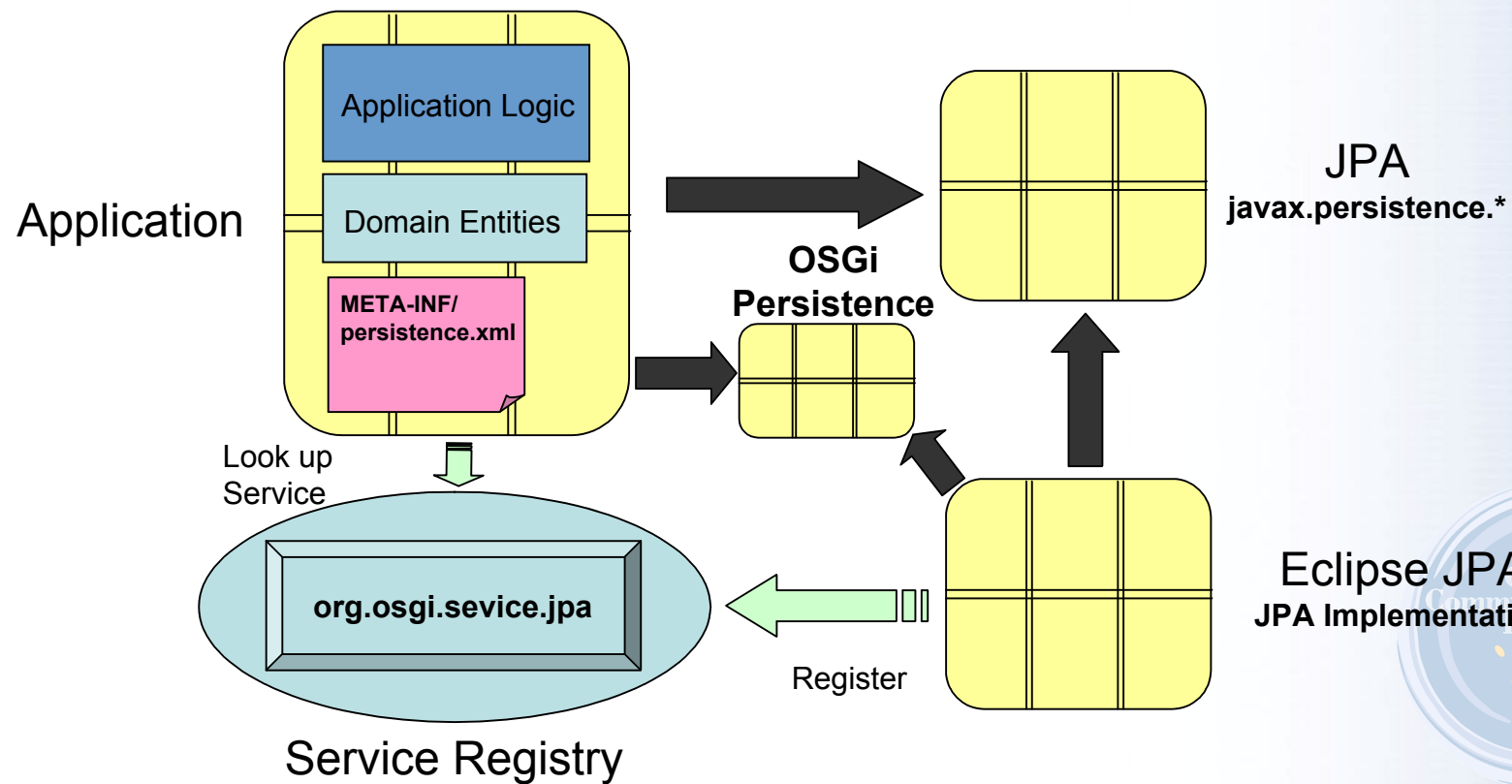


# JPA – The Big Picture



# Intuitive Solution

- Create a service that an application can look up and use to obtain an EntityManagerFactory



Eclipse JPA  
JPA Implementation



## Problems with the Intuitive Solution

---

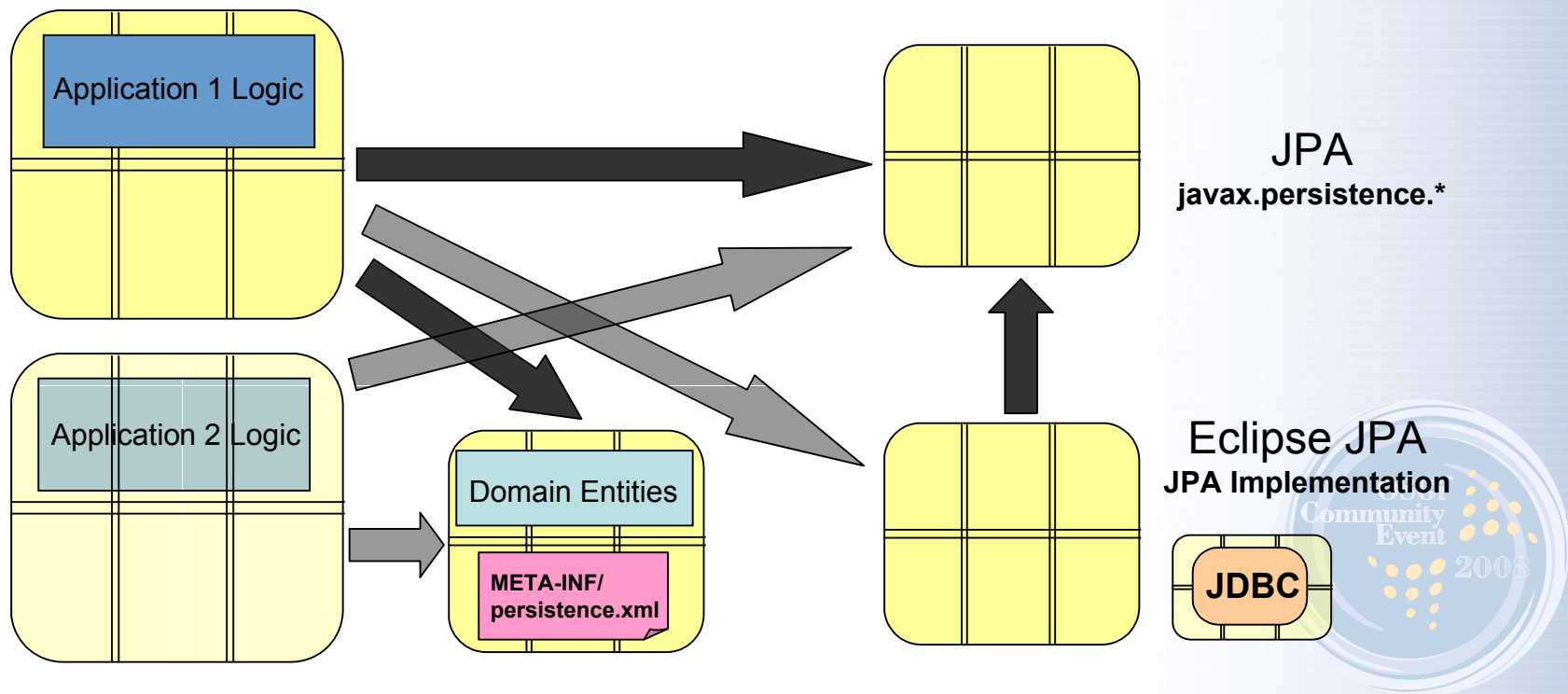
- Need to create yet another interface that does exactly what the existing Persistence class does
- Requires applications to use OSGi-specific service lookups to obtain JPA artifacts
  - More work to bring applications from other envs
  - Not portable across environments
- Forces application to write code to choose specific JPA provider (or not)
  - If JPA provider dependency exists then it is already declared within persistence.xml file => duplication





# Use Cases

- Application bundle + Shared persistence unit bundle + JPA bundle + JPA provider bundle(s)



## Problem: Upward Dependencies

---

- The JPA Provider needs to access resources (persistence.xml and/or other XML files)
- JPA Provider needs to load and introspect domain classes/entities
- List of classes is determined dynamically (by contents of persistence.xml file)
- Classes/resources may be in application bundle, or separate bundle the application depends upon
- Application code calls out to JPA, but not necessarily as part of initialization code



## “Buddy Loader” Strategy

---

- Equinox has “buddy loader” to allow bundles to see up into bundles that depend on it
- Dependent bundle must declare the set of bundles that are its “buddies”
- The application does not usually depend on the underlying provider, only the API
- Not standard, so not portable



## Solution: Standard BundleListener Trick

---

- Introduce a new bundle header:  
“JPA-PersistenceUnits”
  - Applications must add this to specify they are a client of one or more persistence units
- Eclipse JPA registers a BundleListener and keeps track of bundles that start up
- When a JPA client bundle starts, we remember the persistence unit it uses and its classloader
- When client invokes JPA then we use the classloader registered against that client



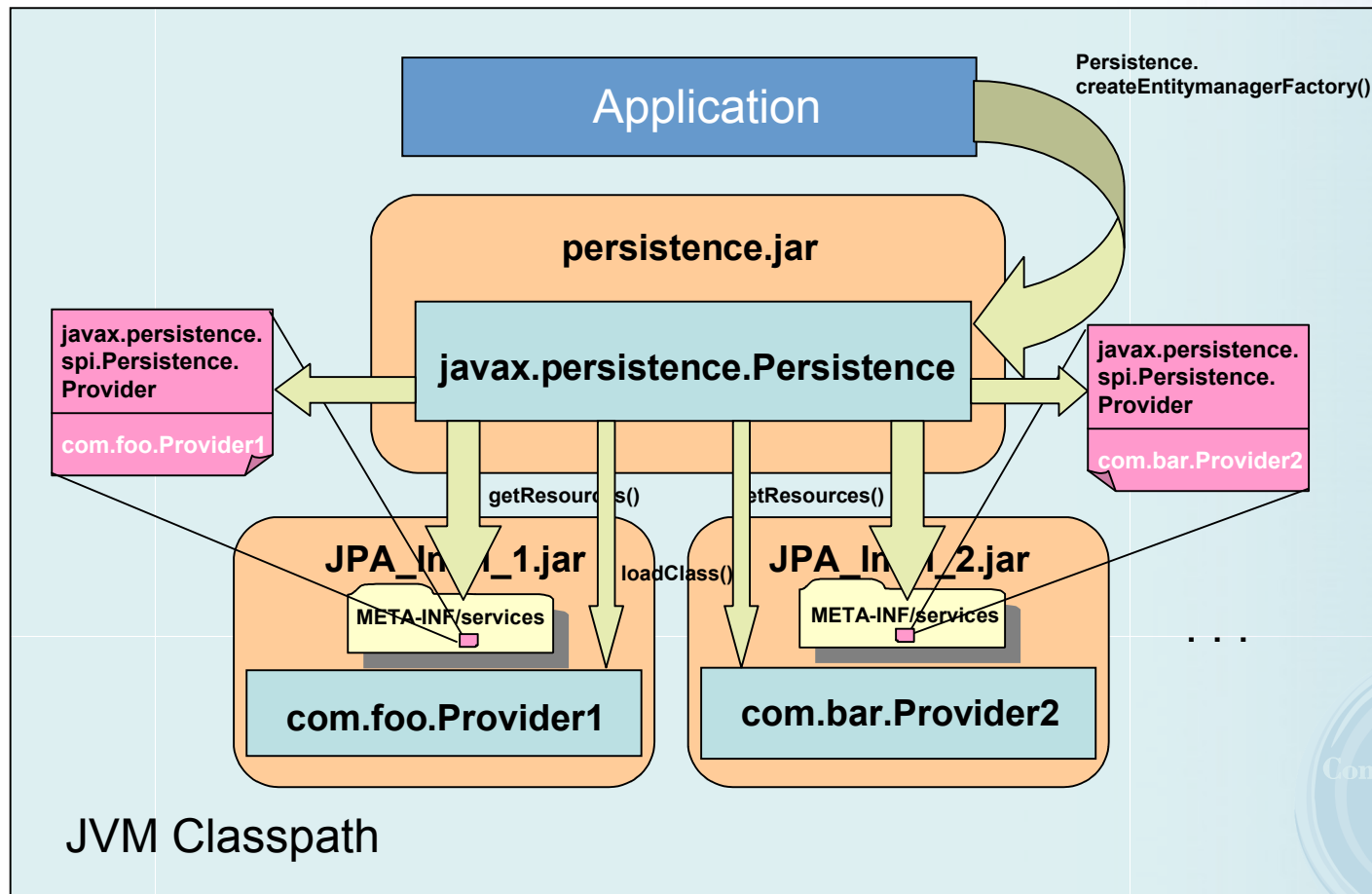
## Problem: The JPA Specification

---

- *javax.persistence.Persistence* is a concrete implementation class in the spec
- Makes use of a service provider pattern documented in JAR specification
- Dynamically looks up service implementations as resources in META-INF/services directory
- Uses context classloader for resource lookup
- Persistence class shipped in JPA spec jar (persistence.jar)



# Service Provider Pattern



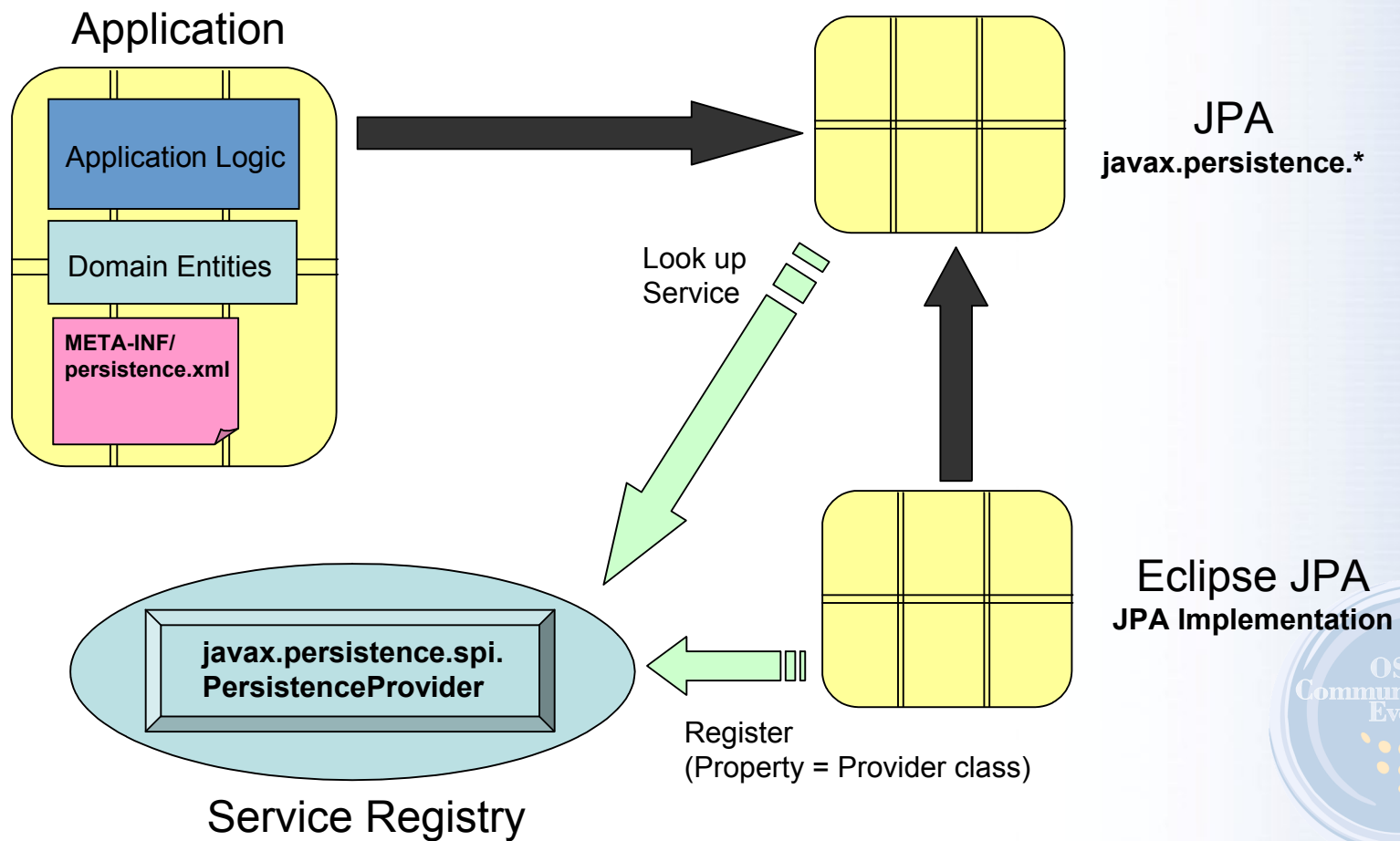
## Solution: JPA Provider Service

---

- Use OSGi service model to allow providers to register themselves as a JPA Provider service
- Allows providers to be dynamically installed and removed
- Pluggable strategy for Persistence class to look up installed JPA provider services
- Persistence class calls out to JPA Provider services to resolve provider
- Hard dependency on a specific JPA Provider averts lookup



# Solution: JPA Provider Service



## Problem: Weaving Domain Classes

---

- Many providers (including Eclipse JPA) employ dynamic weaving techniques
  - Lazy loading, change tracking, serialization issues
- Normally make use of SPI hook in Container env, or Instrumentation (javaagent) in SE env
  - No portable way to achieve this in OSGi environment
- Application code likely causes domain classes to be loaded before calling out to JPA
  - JPA provider “arrives too late to the party”



## Solution: Use Adapter Hooks

---

- Equinox “adapter hooks” provide a similar ability
- Can install an adapter class into the system bundle
  - Will get a callback when classes gets loaded
- Persistence units register a weaving service to indicate they want a chance to weave
  - Adapter class will find the service and pass newly loaded classes to the weaving service to be woven
- Tightly coupled to Equinox
  - Looking at complements in other implementations
- Similar idea to what AspectJ project uses (thanks guys!)



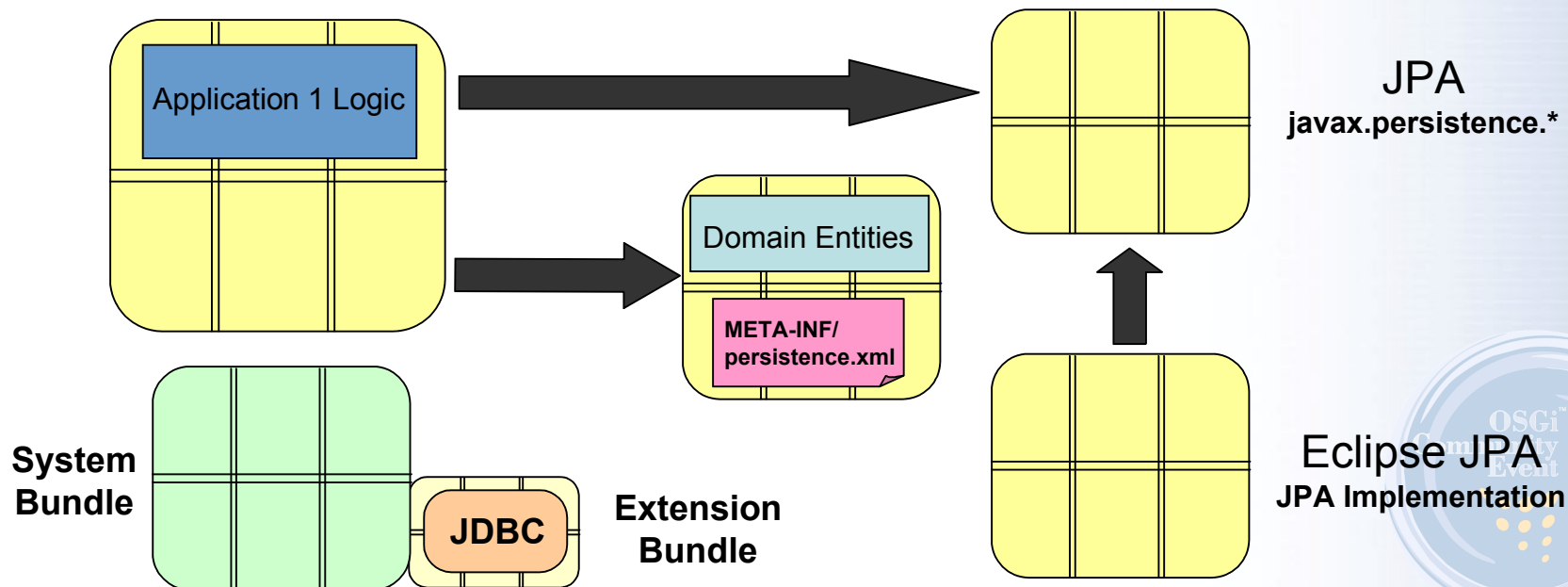
## Problem: JDBC Placement

---

- JDBC driver(s) must be deployed *somewhere*
- Client specifies the driver in its persistence metadata (persistence.xml)
- It is the JPA Provider, not the client, that actually loads and invokes the JDBC classes
- JPA Provider doesn't know *a priori* the driver or data source associated with a persistence unit
- In other environments JDBC driver is usually just available in shared server layer (JAR in lib dir)

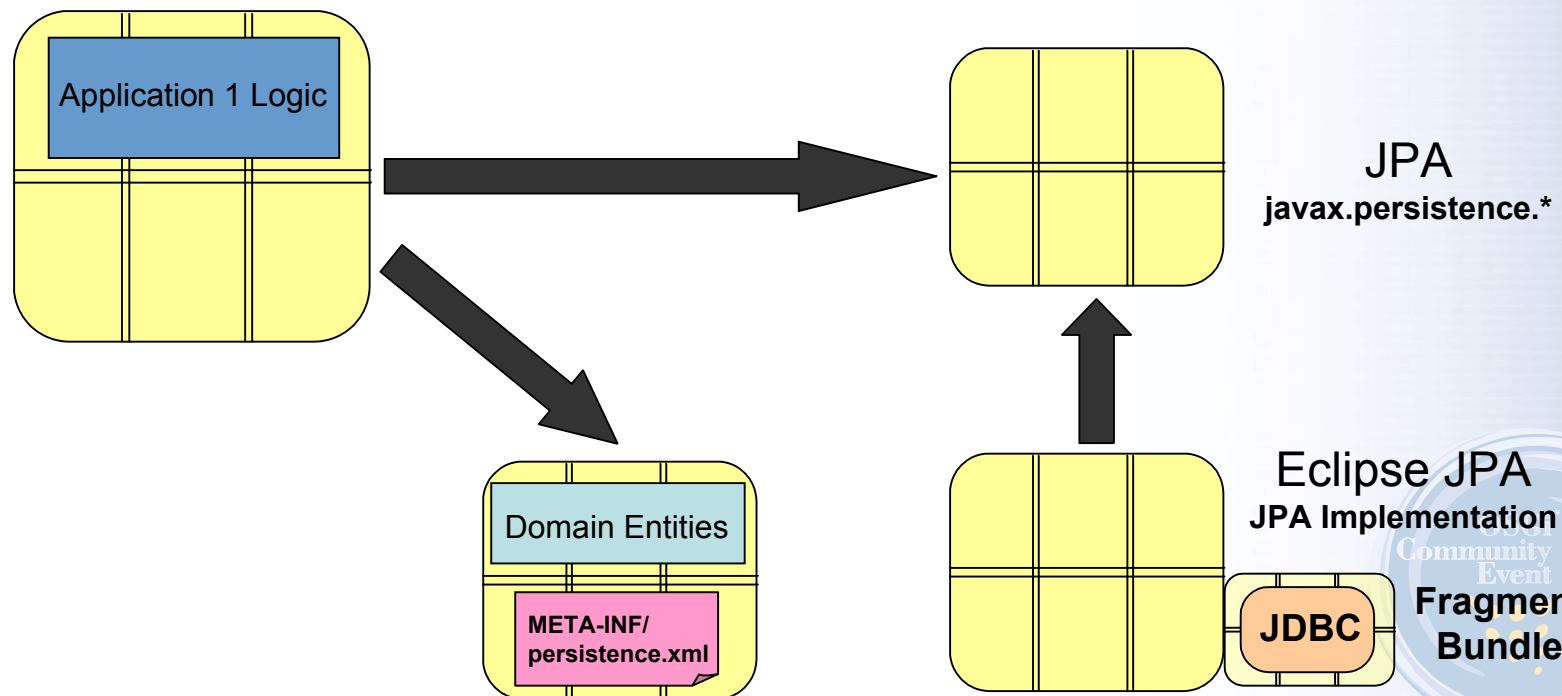
# Provider Extension Bundle

- Add as an extension to the boot classpath
- Refresh requires a framework restart
- Not widely supported anyway



## Provider Fragment Bundle

- Added as a fragment bundle to the provider
- Can't change JDBC version without refreshing provider



## DataSource Property

---

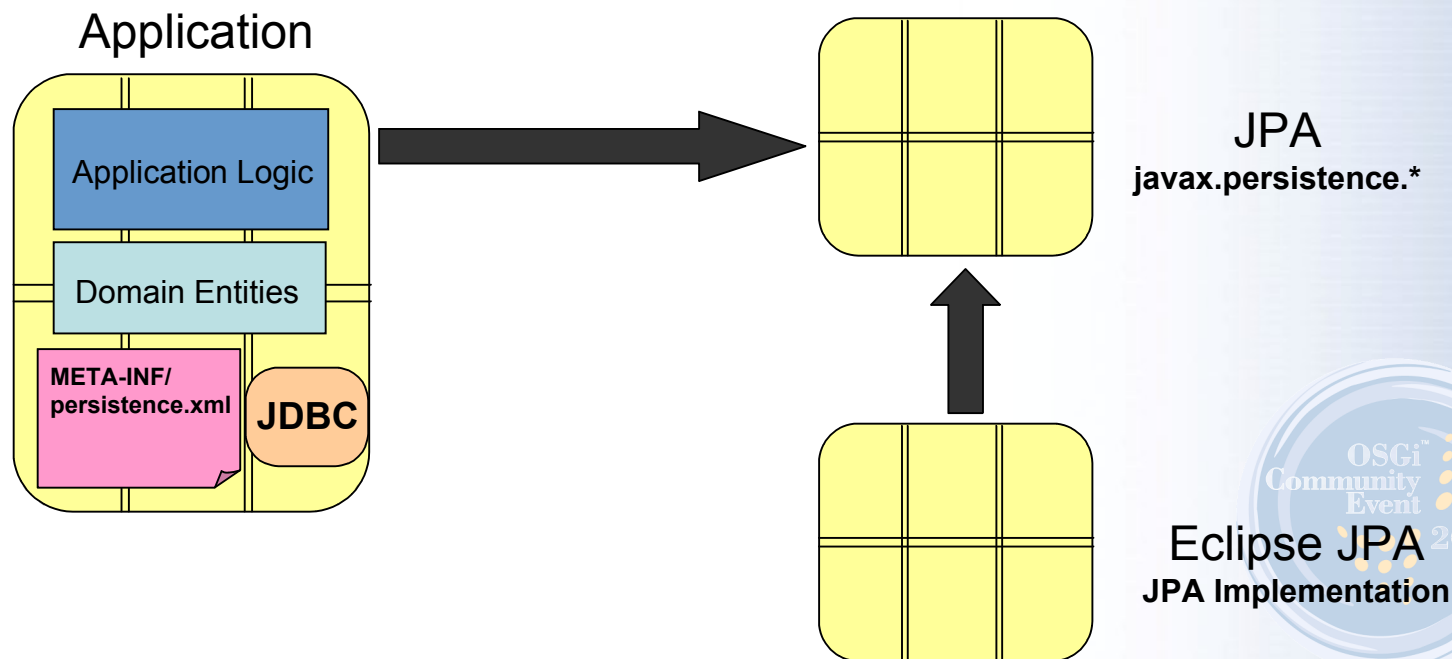
- Persistence.createEntityManagerFactory() facilitates passing an argument Map of props
  - May pass objects (not just String properties)
- Create a data source property that passes the JDBC DataSource to the JPA Provider
  - Need to modify the client to obtain and pass the data source when obtaining the EntityManagerFactory

```
Map props = new HashMap();  
props.put(DATA_SOURCE_PROPERTY, myDataSource);  
EntityManagerFactory emf = Persistence  
    .createEntityManagerFactory("myPUnit", props);
```



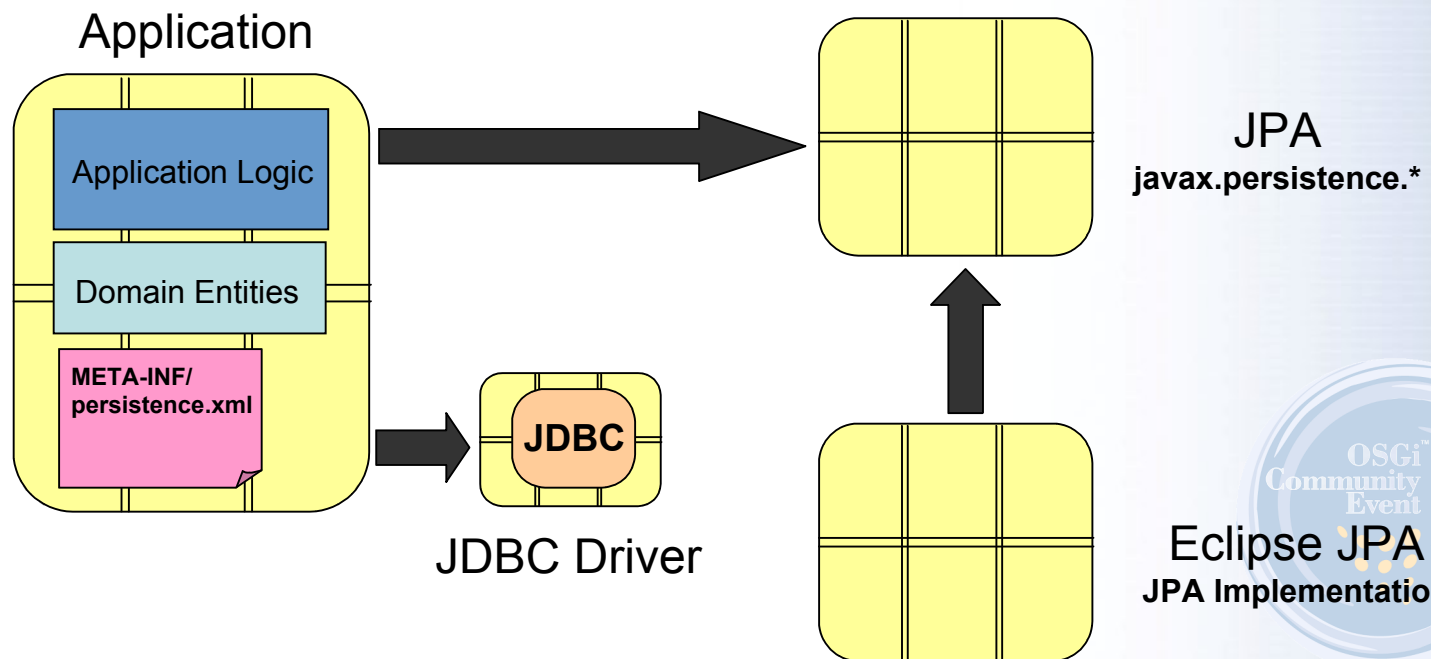
## JDBC Bundled with the Client

- Bundled as part of the client or as a dependent bundle
- Accessed by the provider by virtue of its having the client classloader



## JDBC as a Client Dependent Bundle

- Can independently deploy or upgrade JDBC driver
- Accessed by the provider by virtue of its having the client classloader



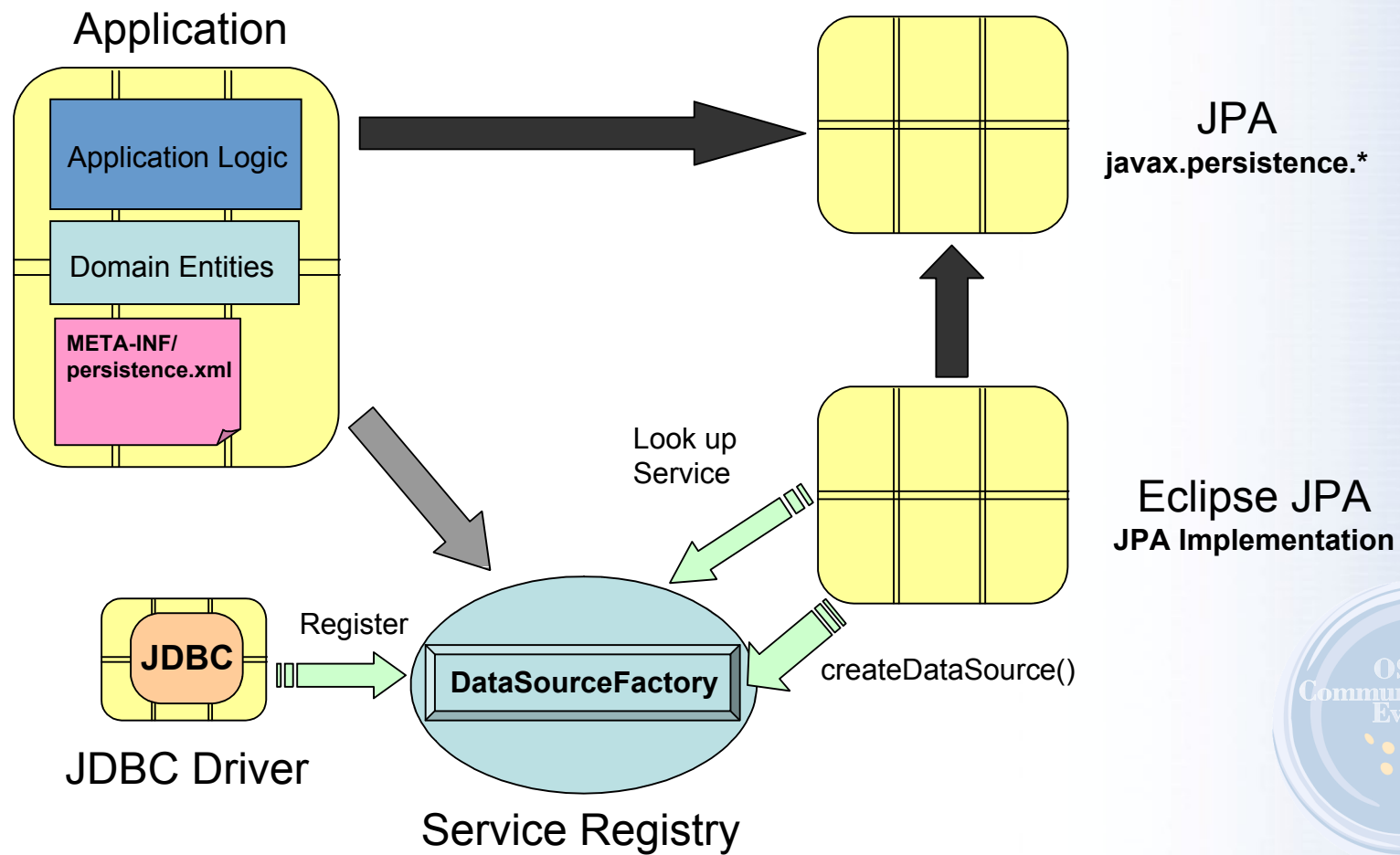
## Solution: JDBC Service

---

- Install as a bundle and register as a service
- Export a DataSourceFactory API that Provider can look up based on driver type
- Achieves bundle independence with a service-based access policy
- Solution designed to be easily extended for other drivers
  - Some standardization of properties is needed
- Generalized solution for JDBC access



# JDBC Service



## Introspection and Questions

---

- Are the issues we encountered integrating JPA with OSGi typical? experiences with other OSGi services?
- Are there fundamental problems with OSGi when it comes to implementing “real” services?
- Are there fundamental problems with JPA that make it less service-based and more application-bound in nature?
- Does JPA need to evolve, or OSGi, or both?



## Future

---

- Look at how hard it would be to support a managed (Container) API
  - Injection, JTA transactions, auto-entity detection, etc.
- Come up with a better classloader strategy?
  - Should an app need to indicate the persistence units it uses in its manifest?
- Sort out the weaving story
- Fix JPA spec to accommodate different provider resolution strategies



## Summary

---

- No standard Persistence service exists, even though persistence is critical to every application
- Are still some general issues that we need to resolve within the context of the specification
- We can work around some of the problems, but can't be implementation-independent right now
- Are anxious to help reify how Persistence services work within OSGi
- Are open to suggestions and offers of assistance and participation from interested parties

