RFC 0143 JPA Integration

Draft

32 Pages

Abstract

The OSGi platform provides an attractive foundation for building enterprise applications, however it lacks support for installing and using standard Java persistence bundles. This RFC describes the features and solutions required to support the Java Persistence API (JPA) in an OSGi framework, both from the perspective of a JPA persistence provider and a JPA user. The different modes, i.e., non-managed or container (managed) mode, of JPA usage are also discussed in this document.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	05/03/09	Initial Draft Mike Keith, Oracle Corporation michael.keith@oracle.com
0.9	05/14/09	Added container-managed sections Mike Keith, Oracle Corporation michael.keith@oracle.com Graham Charters, IBM charters@uk.ibm.com Tim Ward, IBM timothy.ward@uk.ibm.com
0.91	07/23/09	Rewrote much of the document to better align managed and non-managed modes and reuse more of the extender code. Added in the new OSGiPersistenceUnitInfo interface Also incorporated the ideas and discussions from several RFC 143 conference calls. Mike Keith, Oracle Corporation michael.keith@oracle.com Tim Ward, IBM timothy.ward@uk.ibm.com
0.92	08/20/09	Filled in config admin names and removed open issues Mike Keith, Oracle Corporation michael.keith@oracle.com

1 Introduction

Enterprise applications most often end up either reading from or writing to a relational database, and although JDBC is a viable and common means of doing so, mapping objects to the database through an object-relational mapping (ORM) library is often preferred. Thus, before the goal of developing enterprise applications in OSGi can be realized there must be a way for ORM to be used within an OSGi framework. JPA provides a standard way of mapping Java objects to a relational database, as well as a runtime API and query facility for loading and storing objects on demand. As the standard for enterprise persistence, being supported by all major middleware vendors, JPA is the natural choice as the vehicle to offer ORM persistence support in OSGi.

This RFC describes solutions to the problems that surround running and invoking a JPA implementation in OSGi. Specific features of OSGi and JPA are referenced and leveraged. It is assumed the reader is familiar with the OSGi concepts and somewhat aware of the JPA concepts, although some brief definitions are provided in the terminology section.

2 Application Domain

The problems that emerge when using JPA in OSGi can be viewed from 3 different perspectives: the standard JPA interfaces, the JPA persistence provider (implementation classes), and the application. Each has a runtime role to play and each has the potential to produce repercussions on the classloading and resource access.

JPA may execute in one of two separate and somewhat different runtime contexts. The first is a simple Java SE environment, where the classpath is a JVM-level configuration setting and the application gets control once the primary JVM bootstrapping code has completed. The second is a JPA Container environment (e.g. Java EE server) in which application code is invoked only as a result of a client request. Each has different conditions, startup sequences, APIs, code paths, and problems, so each must be considered individually.

This RFC either relies upon or leverages the following existing RFCs:

RFC 122 – JDBC – Access to the database

RFC 98 – Transactions – Integration with JTA

RFC 124 – Injection of JPA resources into components

2.1 Persistence Descriptor

The standard unit of persistence configuration is called the *persistence unit*. One or more persistence units are configured in an XML persistence descriptor file called *persistence.xml*. The persistence.xml file must be in the META-INF folder of a root folder or jar file that is on the classpath or otherwise accessible to the relevant classloader.

Various facets of the persistence unit configuration are specified in the persistence.xml entry for that persistence unit, with some aspects applying to Java SE execution, and others only relevant in a container execution context, still others valid in both contexts. JPA developers are expected to be familiar with the elements that are applicable to the context they are running in.

2.2 Non-Managed Environment

In standalone Java SE environments the API and the provider implementation jar(s) are the only additional classes that must be present on the classpath. The standard API classes may be included in the provider jar, but are typically in a separate jar file on the classpath.

2.2.1 Bootstrapping

A client application would use the Persistence class to bootstrap the persistence provider and obtain an EntityManagerFactory the following way:

EntityManagerFactory emf = Persistence.createEntityManagerFactory(“Accounting”);

The causes the prescribed bootstrap control flow:

1. Persistence class uses the service provider mechanism to find all available providers (it looks on the classpath for **META-INF/services/javax.persistence.spi.PersistenceProvider** files and finds the implementing classes in the file contents)

2. Persistence class asks each provider in turn to try to create an EMF for the given persistence unit name until one is successfully returned (or it throws an exception)

3. A provider is expected to look for all META-INF/persistence.xml files on the application classpath and process the files, looking for the named persistence unit to see if its own provider class is specified as the provider for that persistence unit

4. If the persistence unit does not specify a provider class, or specifies one that identifies the provider that is currently looking at the persistence unit, then that provider will create an EMF and return it to the Persistence class. If not then it will return null.

5. Once the Persistence class returns the EMF to the calling client the EMF is invoked to obtain an EM, and client persistence operations may be performed from that point on

As expected, the classloader has a great deal to do with how things get loaded at the outset, and can be the source of many bootstrapping problems when the class dependencies are neither defined nor possibly even known. Additionally, due to the multitude of possible deployment configurations, such as a client jar (bundle) having dependencies on the domain classes that are in a separate jar (bundle), it is even more difficult to quantify and organize those dependencies.

2.2.2 Accessing Domain Classes

When a provider creates an EMF one of the things that it must do is create a set of internal metadata structures that it uses to control the persisting and loading of the persistent entities defined in the persistent unit. This metadata is typically obtained by scanning the annotations on the entity classes and determining the mappings and entity class meta-information. The classes are known because they are explicitly listed in the persistence.xml file, but obtaining the metadata and structure from them involves doing one of two things:

1. Accessing the classfiles and using bytecode analysis

2. Loading the classes and using the reflection API on them

In either case the provider must be able to locate the domain classes. Once the class metadata is known, the provider can piece together the metadata for all of the persistence unit.

Domain classes may also be contained within JAR files that are referenced by location path in the persistence.xml file. The JAR files must be searched for classes annotated with JPA-denoting annotations.

2.2.3 XML Mapping Descriptors

A provider may be required to read and process one or more XML mapping descriptor files instead of, or in addition to, processing annotations. A default mapping file must be searched for, and zero or more other mapping files listed in the persistence.xml file need to be processed.

A mapping file contains additional mapping metadata for managed classes in a persistence unit. Mapping files must be entirely processed before the metadata for the persistence unit can be completely composed.

2.2.4 Class Weaving or Proxying

An important feature provided by JPA is that of lazy loading, or the ability to defer loading a related object or a part of the state of the original object. The developer of the persistent class declaratively denotes specific state as being lazily loaded, and the provider must do its best to only load that state on demand. Lazy loading is primarily a hint in that the provider is not making any guarantees, but most providers fully support lazy loading and most users would notice the performance degradation if lazy loading were to be unavailable.

Lazy loading of collection-valued relationships can actually be done simply by the provider substituting a provider-specific collection instance (that happens to implement the Collection interface) in place of the real one. This allows the provider to intercept calls to access the related objects and fault them in on demand. With single-valued relationships, however, either weaving or proxying of the class is necessary, and to support lazy loading of basic entity state, weaving or bytecode enhancement in some form is required.

Dynamic weaving is the practice of changing the bytecodes of a class at the time the class is loaded. Weaving may be done statically, requiring an additional post-compilation step that generates a separate set of classfiles, but developers tend to shrink away from the static approach, instead favoring the transparent dynamic weaving that does all of the manipulation at runtime and does not involve any additional development phases or artifacts.

Proxying is the technique whereby a dynamic proxy subclass is generated and used in place of the original. The proxy subclass instance inherits the state and behavior from the original class, but offers the proxy implementation an opportunity to intercept the method calls. Proxies may refer to other proxies, managing lazy relationships or references to other entities, but proxies can't offer support for lazy loading of primitive fields.

2.2.5 Database Access

JPA accesses the database through JDBC. In a typical SE environment the JDBC driver properties will be specified in the persistence.xml file and the driver will be loaded and invoked directly by the persistence provider. The driver classes are expected to be accessible and on the classpath in the execution environment that is configured by the user.

2.3 Managed Environment

In a Java EE environment a default JPA provider is usually available as a server-level library. In some cases the provider may be included with the deployment archive, such as a WAR, but ultimately the JPA library and a provider must be available to the application.

2.3.1 Management of Entity Managers and Factories

In a managed environment a persistence unit is deployed within a deployment archive such as an EAR, a WAR or an EJB JAR. The container is responsible for the initialization of the persistence units defined within the persistence.xml files in the deployment archive, and will typically manage the entity manager factories and entity managers that are used within the deployment components. Application components will obtain references to entity managers and factories either by looking them up in JNDI, or by injecting them into component fields or properties. The container ensures that at component (EJB) initialization time the component instance is injected with the appropriately configured entity manager or factory instance.

```
@Stateless
public class MyStatelessBean {
    @PersistenceContext(unitName=”Accounting”)
    EntityManager em;

    // …
}
```

Similarly, when the scope of the enclosing component has ended then the managed entity manager must also be cleaned up and discarded by the managing container. The container is thus responsible for the allocation and disposal of managed entity managers and factories.

2.3.2 Propagation

A persistence context represents the collection of entities, within a given persistence unit, that are managed at a given time. When a component is injected with an entity manager and control is transferred to another enterprise component within the same JTA transaction then the persistence context is also propagated to the called component. This is called persistence context propagation, and is possible because the container controls both the components and the entity managers that are being injected.

2.3.3 Service Provider Interface

The container-provider SPI requires that when a container deploys a persistence unit a container-managed factory is created by invoking the `createContainerEntityManagerFactory` method on the provider interface. This is done by reading the persistence.xml files and processing the information in them and passing the information to the provider.

The container is responsible for determining which provider backs a given persistence unit. It does this by:

1. Reading the persistence.xml files and determining if a provider is hard-coded within the persistence units
2. Choosing a default provider based on internal rules if none was specified

The container passes the persistence unit configuration information to the chosen provider through a container-implemented interface argument called a `javax.persistence.spi.PersistenceUnitInfo`. This obviates the need for the provider to discover, read or process the persistence.xml files. The `PersistenceUnitInfo` contains all of the information contained within the persistence.xml files. The information of interest that the container passes to the provider is:

- transaction type (defaults to be a container JTA transaction)
- JNDI name(s) of data source(s) for the provider to connect to
- the file paths of XML mapping files containing object-relational mapping information
- the URLs of additional JAR files that may contain JPA managed classes
- names of managed classes that were specified in the persistence.xml file
- classloader from which the resources and managed classes are visible
- temporary classloader that can be used to load the resources and classes and then be disposed
- URL of persistence root
- means for the provider to add a transforming weaver to the classloader, so that at classload time the transformer can dynamically weave classfiles

2.3.4 Automatic Discovery of Managed Classes

One benefit of container deployment is the feature that the application does not have to list all of the managed classes in the persistence unit configuration. The container/provider combination is expected to be able to

discover and detect all of the JPA entities and managed classes in the deployment unit (and in the jar files listed in the persistence unit) by looking at the annotations on the classes, in addition to the classes that may have been mapped in one or more XML files.

2.3.5 Transaction Integration

The Java Transaction API (JTA) is the transaction system used in Java EE containers, and when running in a managed environment JPA is expected to be integrated with JTA. The provider must be able to enlist in JTA transactions and subscribe to JTA notifications, obliging it to become a party to and a participant in the commit and rollback semantics of JTA.

2.4 Deployment Configurations

One factor that complicates the issues is the way the application is packaged and the bundles are partitioned. The following configurations are possible when using JPA:

2.4.1 Single Client Bundle

The most common scenario is expected to be one in which the provider is configured as one bundle, the persistence API classes in another bundle, and the client packaged as a third bundle, as seen in Figure 1.

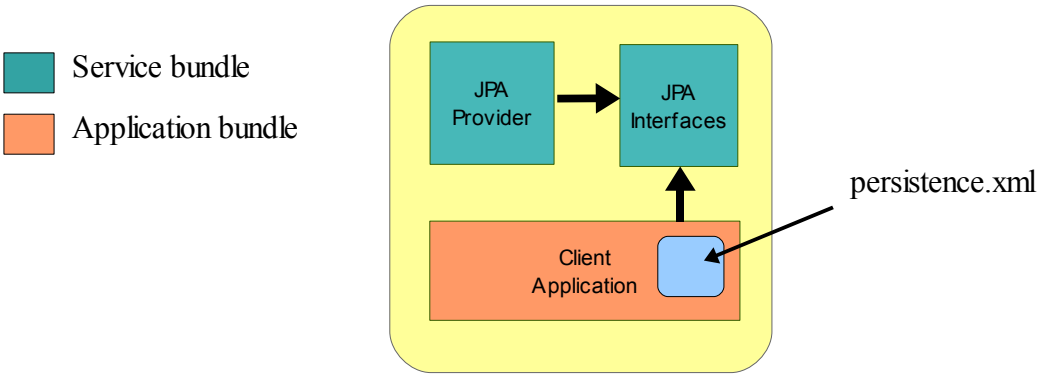

Figure 1. Single Client Bundle Configuration

2.4.2 Combined Provider and API Bundles

In some cases the provider may actually include the Java Persistence API classes as part of its bundle, as in Figure 2. The Persistence implementation must be robust enough to find other providers outside the bundle.

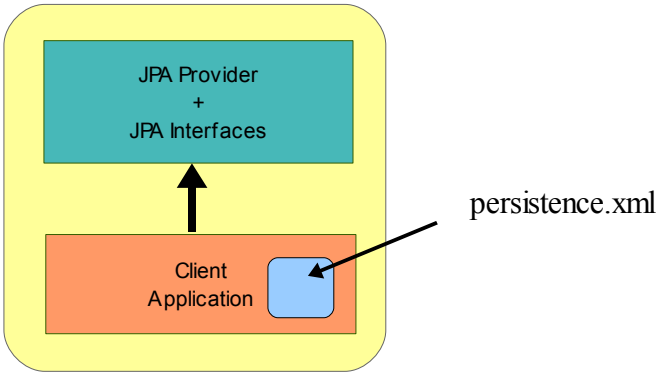

Figure 2. Combined Provider and API Bundles

Providers that want to ship in this configuration will be required to export the JPA packages in order to support multiple versions of JPA. Providers in a container environment.

2.4.3 Driver Bundle

Drivers may be packaged as independent bundles that can be used directly by other applications in addition to being used by JPA. Figure 3 shows the case where the bundle is being specified by the client persistence unit, even though the client does not have any hard compile-time dependencies on the driver. The JPA provider also does not have any hard dependencies upon the driver. However, at runtime, the provider must load the driver specified by the persistence unit.

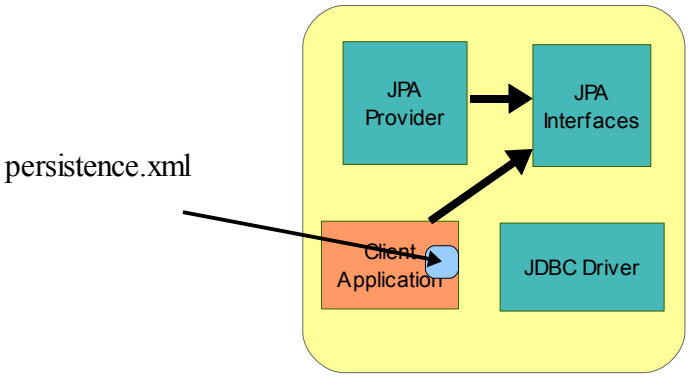

Figure 3. Independent Driver Bundle

2.4.4 Client Bundling of Driver

Drivers may be packaged as part of an application when the driver is not shared, and not OSGi-enabled. Figure 4 shows a client bundle that includes the driver classes in either an enclosed jar file or as classes in the jar. This is equivalent to the driver being in a separate bundle that is a required bundle of the client.

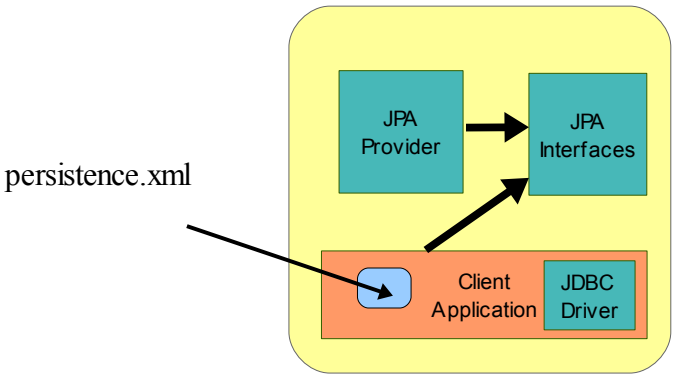

Figure 4. Client Bundle Including Driver Classes

2.4.5 Persistence Unit Dependency

The client may, in fact, be a network of bundles being accessed through various package or even bundle dependencies, and look like either Figure 5, where the persistence unit is actually defined in a separate bundle.

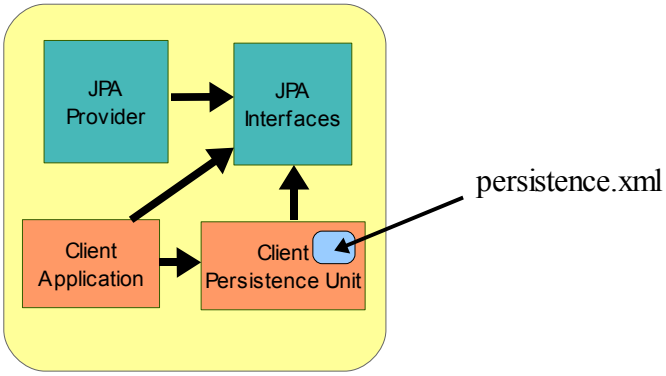

Figure 5. Persistence Unit Dependency

2.4.6 Domain Model Dependency

In this packaging the persistence.xml resource is local, but the managed classes are imported through package dependencies and defined in another bundle. Figure 6 shows that the domain classes may be separate from the client, and hence may not even be aware of their persistence.

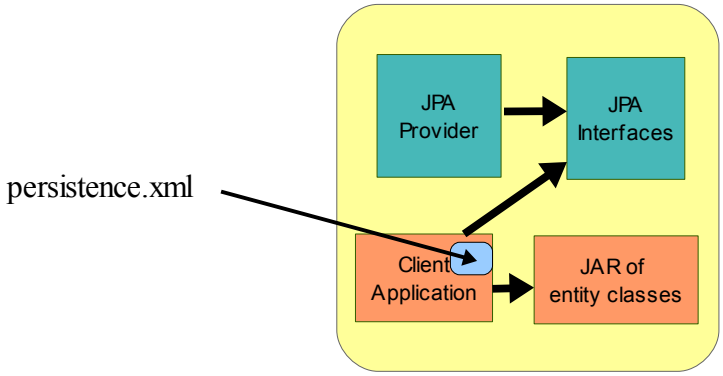

Figure 6. Domain Model Dependency

Support for this scenario will not be required in this release. Vendors may find their own solution, or they may require that it be packaged as a persistence unit, with a persistence.xml file.

2.5 Terminology + Abbreviations

- Application-managed Entity Manager: An Entity Manager that has a life cycle that is controlled by the application
- Container-managed Entity Manager: An Entity Manager that has a life cycle that is controlled by the container
- Entity Manager: The main entry point object to provide JPA operations
- Java EE: Java Enterprise Edition, the enterprise Java development and runtime platform
- Java SE: Java Standard Edition, the standard Java development and runtime platform
- JCP: Java Community Process
- JPA: Java Persistence API
- JPA Container: Any container that supports and implements the container Service Provider Interface (SPI), invokes JPA providers, and injects EntityManager and EntityManagerFactory instances into client classes
- JPA Interfaces: The set of Java classes and interfaces defined by the JPA specification, including 1) the user-level interfaces, 2) the SPI interfaces between the container and the provider, and 3) the Persistence bootstrap class and other classes that are defined and must be implemented by proivders
- JPA Provider: A vendor implementation of the JPA interfaces
- Managed Class: Class that can be mapped to the database, viz. an entity, embeddable or mapped superclass
- Managed Environment: Mode in which a Container or component manager is resident and interfacing with JPA. Applications obtain an EntityManagerFactory or EntityManager by injection or by lookup (e.g JNDI)
- Non-managed Environment: Mode in which there is no container or managing layer and the application uses the Persistence bootstrap class to obtain an EntityManagerFactory and then EntityManager
- SPI: Service Provider Interface, or an interface that can be used to invoke a JPA Provider offering a specific service
- Persistence Provider: see JPA Provider
- Persistence Unit: The JPA unit of configuration stored in an external persistence.xml file

3 Problem Description

3.1 Non-Managed Environment

The following sections describe the JPA issues in a typical non-managed or traditional Java SE environment.

3.1.1 Provider Discovery

The discovery of JPA providers by the Persistence class in OSGi is a difficult task when using the traditional existing approach of looking for resources on the classpath because the classloader simply does not have the visibility to find all available providers in the system. The Persistence class neither knows nor cares which providers are actually loaded in the framework so it should not need to have any dependency upon particular provider implementation classes, especially given that providers may come and go in an OSGi world. Neither can the Persistence class have a simple package-level dependency upon the provider SPI interface, since a persistence unit may specify a particular concrete provider, thus it would not be appropriate for the framework to wire the SPI package to a random provider.

3.1.2 Finding and Processing Persistence.xml

When the provider is invoked by the Persistence class to create an EMF the only way that it knows whether it can and should create an EMF is by looking for the persistence unit definition within some persistence.xml file on the classpath. The problem is that the persistence.xml file is actually going to be on the client classpath and the provider will certainly not have any dependency on a client or persistence unit bundle. To make matters worse, the provider does not even know which client is asking for the persistence unit, so it would appear to be forced to go through every bundle in the framework looking for the persistence.xml file that has that particular named persistence unit in it.

A peripheral issue related to the locating of a persistence.xml file is that it is a resource and is looked for as a resource. Because of this, it need only be on the classpath of the client, without any additional JPA metadata to describe or specify it. In OSGi there should be some way to depend upon the resource, without necessarily having to declare a bundle dependency upon the bundle in which the resource is located.

3.1.3 Driver Access

To complete the initialization of a persistence unit the provider must establish at least one connection to the database through the JDBC driver that is declared within the persistence.xml file. The provider is decoupled from the driver in that it cannot know *a priori* which driver will be required by a given persistence unit until it processes the file at runtime. The provider cannot either import the driver packages, or define any bundle dependencies, but must find a way to load the driver class on demand when a persistence unit is loaded.

3.1.4 Loading Managed Classes

A similar problem exists for domain classes as for driver classes, but from a slightly different perspective, and with slightly more challenging circumstances. The provider only knows the entity, embeddable and mapped superclass classes once it processes the persistence.xml file. The provider needs to load the classes in order to analyze them and determine the mappings and annotations on them, as well as introspect them to come up with defaults in the case where no mappings are specified.

Once again, however, the domain classes are not on the classpath of the provider. They may be in any one of the client bundles described in the use cases in section 2.4. The provider must obtain the correct loader to allow it to load the managed classes.

3.1.5 Classloading Order

A provider may choose to use weaving, code generation, reflection, or some combination of these. Dynamic weaving of the entity classes means that the provider modifies the class at class-load time in order to install transparent persistence features like change tracking and lazy loading into them. Even if the provider can load the managed classes, it may be too late to perform such weaving, since once the classes have been loaded the window for weaving has already closed. Dynamic weaving of a class within a given classloader can only be performed the first time that class is loaded within the particular loader. That presents a rather difficult challenge to JPA providers because in OSGi when a bundle loader loads classes contained in that bundle there is no way for other bundles to have an opportunity to intercept the classloading process. A JPA provider doesn't even necessarily know the classes that need to be woven until it has processed the persistence unit, which possibly occurred even after the the classes were already loaded.

3.1.6 Dependencies Introduced by Weaving

In order to make use of Managed Classes, JPA providers “enhance” the class bytecode in a provider specific way, either by directly modifying the class or by subclassing. The JPA specification allows for this “enhancement” to occur at class load time, however this causes problems in an OSGi environment. The enhancement process can cause a managed class to become dependent on provider-specific interfaces and classes, but unless the bundle containing the managed classes expresses this dependency, after enhancement these classes will not be loadable by the bundle's class loader.

3.2 Managed Environment

The container environment poses many of the same problems that were described in the non-container environment, a main difference being that the container, instead of the Persistence class, is often in the position of figuring things out. A few additional problems are also present, though, because of the fact that there is an SPI between the container and the provider.

3.2.1 Provider Discovery

A container will often choose a default provider based on its own preferences, for example a specific container will ship with a preconfigured provider that ships with the container and may even be implemented by the same vendor. If this situation does not apply to a given container then it will be in the same position as the Persistence class is in non-managed scenario, and will need to discover the various providers that are active.

3.2.2 Finding and Processing Persistence.xml

The container must find, read and process the persistence.xml files, so depending upon its notion of a deployed application it may or may not have access to internal mechanisms that provide classloading access to the deployed bundles.

3.2.3 Driver Access

In the non-managed case there is no preconfigured data source, but in a typical container environment data sources are configured to be present in JNDI and looked up when needed. If we assume that the container will indeed support the installation of a transactional JTA data source in JNDI then the problem will only be that the provider must be able to access a JNDI context that contains the data source. If the container does not offer the traditional data source service of a JTA-enabled data source, then we must solve the problem of getting the data source obtained from RFC 122 JDBC drivers to enlist in JTA, and find a way for the provider to obtain access to that data source.

3.2.4 Loading Managed Classes

Even though it is the container deployment that enables automatic entity discovery, it is still the provider that discovers the classes. It does so using the URL and typically the temporary classloader that it obtains from the `PersistenceUnitInfo`. The container must be able to pass the correct classloader and temp loader to the provider so that the provider can access all of the resources and classes. The difficulty in doing this may be sever, or it may be simple, depending upon the container, however, if we assume that the container does have some kind of control or access to the deployment artifacts then it should not be onerous.

If the container does not control the artifacts in the same way that Java EE does then it will have the same problems as the Persistence class does in having no way to get at the correct classloader for the domain classes.

3.2.5 Class Transformation (Weaving)

The container must have a direct hook to the application classloader to enable it to support the addTransformer method to the provider. Unless the container is tightly integrated with the environment, i.e. not only privy but providing its own classloading layer to the various deployment bundles, then it will not be able to make good on the container transformation promise, since transformation will require the ability to intercept the classloading process and replace the loaded class with the result from the transformer.

The same problem of package dependencies introduced by weaving, described in the non-weaving process, also exists in the managed scenario.

3.2.6 JTA Transactions

The provider must have a way to access the JTA transaction manager or equivalent Java EE TransactionSynchronizationRegistry object. For the provider to enlist in the current JTA transaction and be synchronized with respect to that transaction, it needs access to a transaction artifact that supports the necessary operations of adding synchronization listeners and obtaining the transaction status. This will allow it to participate and be atomic with respect to JTA.

4 Requirements

1. A bundle MUST be able to be compiled without having to import the JPA provider implementation packages.

2. Extra metadata MAY be required in the calling bundle manifest to indicate the persistence unit(s) used by the bundle, independent of the persistence.xml file that it may be defined in.
3. It SHOULD be possible to import a JPA provider that is bundled together with the JPA interfaces.
4. It SHOULD be possible for an application to bundle the JDBC driver used by its persistence unit.
5. It MUST be possible for an application to import and access provider-specific classes.
6. Existing application code SHOULD NOT need to be aware of the OSGi framework in order to continue to use JPA within an OSGi environment.
7. Applications MAY contain vendor dependencies either by specifying a particular provider class and/or by importing value-add provider packages.
8. The service registry MAY be used to publish the existence of a JPA provider or to look up providers that are available.
9. A bundle SHOULD be able to use a persistence unit defined in a separate bundle, although this MAY require creating a bundle dependency.
10. The design SHOULD NOT render any particular strategy or JPA implementation choice impossible to be invoked in an OSGi environment.
11. It SHOULD be possible for a JPA client bundle to remain installed while the version of the provider is upgraded. It may not be possible if the provider does class generation or class modification that produces different results in each of the versions.
12. Container-managed entity managers MAY only be available if a suitable JPA container is active.
13. It MUST be possible to support JTA entity managers, although they may only be supported in container mode.
14. Non-container JPA usage MAY only support local transaction usage.
15. An OSGi-compliant JPA persistence provider MUST NOT be impeded from also being compliant with the JPA specification.
16. An OSGi JPA persistence provider MAY provide additional aspects of the technology that are required for JPA to be properly integrated in an OSGi framework but MUST NOT make any syntactic changes to the Java interfaces defined by JPA.
17. The OSGi JPA persistence provider design MUST NOT require an Execution Environment greater than that which satisfies the signatures of JPA.
18. A JPA application that works in OSGi MUST also work outside of OSGi.
19. It MUST be possible to have multiple active persistence providers in the same OSGi Framework instance.
20. It MUST be possible to have multiple versions of the same persistence provider coexist in the same OSGi Framework instance.

5 Technical Solution

The most appropriate mechanism for overcoming many of the issues described in the previous section is through the use of an extender. The extender can do the necessary bundle snooping to discover the provider and application bundles and react accordingly. The following sections describe the details of the JPA extender and its responsibilities, as well as the roles that must be played by the Container, the provider and all of the other agents in the system.

5.1 Provider Service

JPA providers may come and go over time, with multiple ones active at any given point. As each one is detected it is registered as a service in the Service Registry.

5.1.1 Registering as a Service

Providers are registered in the service registry under the JPA provider SPI class, `javax.persistence.spi.PersistenceProvider`. It is assumed that each provider will have a services file, `META-INF/services/javax.persistence.spi.PersistenceProvider`. According to the JPA spec providers should be including this file in order to be detected. When a JPA provider bundle is started it will be detected by the extender and registered in the service registry.

The extender must read the contents of the services file and use the class name contained there for two purposes:

1) It must instantiate the class and pass it as the implementation for the PersistenceProvider service. This hooks the provider class into the registry and allows it to be directly invoked.

2) When registering the provider service the extender must also specify a `osgi.jpa.provider.name` service property, supplying the name of the provider implementation class as the value. This enables a specific provider to be directly located and used in the event of a provider class being specified as the value of the standard `javax.persistence.provider` JPA property in the `createEntityManagerFactory` method, or in a `provider` element in the persistence.xml file.

5.1.2 Provider Versioning

JPA does not define any notion of provider versioning. To enable differentiation of provider versions running in the same framework instance, a service property named `osgi.jpa.provider.version` may be optionally included during provider service registration. The extender must set the value of this property to be the value of the Bundle-Version manifest header from the provider bundle.

5.1.2.1 Dependency on a Provider Version

A persistence unit may have a non-code dependency on a version of a provider and can specify the use of a specific provider version as a persistence unit property named `osgi.jpa.provider.version`. The syntax of the property value is the same OSGi version range syntax used for Import-Package or Require-Bundle.

Example:

```
<persistence>
```

```
  <persistence-unit name="CustomerInfo">
    <provider>com.acme.PersistenceProvider</provider>
      ...
    <properties>
      <property name="osgi.jpa.provider.version" value="[1.0.1,1.1.0)" />
    </properties>
  </persistence-unit>
</persistence>
```

5.1.3 Discovering Providers

In JPA 1.0, providers were discovered by being on the classpath of the application or calling bundle. In OSGi we must instead do a lookup in the service registry for provider services, using the `osgi.jpa.provider.name` provider name property as a filter.

In JPA 2.0, there is a provider resolution method that allows a resolver strategy to be plugged in and used by the Persistence class when it needs to resolve the providers. This is also used by the Persistence.getPersistenceUtil().isLoaded method.

The resolver class must implement the PersistenceProviderResolver interface, and would look up the Provider SPI interface from the Service Registry to obtain the list of providers that have been registered.

Example:

```
public class OSGiProviderResolver implements PersistenceProviderResolver {

    public List<PersistenceProvider> getPersistenceProviders() {

        List<PersistenceProvider> providers = new ArrayList<PersistenceProvider>();
        ServiceReference[] refs = null;
        try {
            refs = ctx.getServiceReferences(
                            "javax.persistence.spi.PersistenceProvider", null);
        } catch (InvalidSyntaxException invEx) {} // Can't happen (filter is null)
        if (refs != null) {
            for (ServiceReference ref : refs) {
                providers.add((PersistenceProvider)ctx.getService(ref));
            }
        }
        return providers;
    }

    public void clearCachedProviders() {}
}
```

The resolver may cache the providers to make validation and other uses of isLoaded() not be expensive method calls, so when a dynamic resolution of providers is required the cache must be cleared beforehand. The cache may also be cleared when it is known that a provider is being stopped.

The resolver class must be set in the resolver holder using the setPersistenceProviderResolver method:

```
PersistenceProviderResolverHolder.setPersistenceProviderResolver(
                                                new OSGiProviderResolver());
```

5.2 Packaging of JPA Interface Classes

The JPA interfaces will be expected to be packaged in their own bundle, separate from the vendor classes. This will allow all applications with similar JPA version requirements to resolve to the same Persistence (and PersistenceProviderResolver) classes. However, if one or more vendor providers decides to package the API classes as part of the provider bundle then a client may get wired to the API classes in that bundle, causing the application to access a different Persistence class implementation. Providers that package the standard JPA interfaces/classes must export them appropriately.

5.3 Persistence Unit Service

Each persistence unit is registered as a service in the service registry.

5.3.1 Persistence Unit Detection

The extender becomes aware of persistence unit definitions, typically by detecting the presence of one or more persistence.xml files. If a persistence.xml file is contained in the META-INF directory of the root of the bundle, the root of any jar file on the Bundle-Classpath, the WEB-INF/classes directory of the bundle, or in the root of any jar in the WEB-INF/lib directory, then the bundle is assumed to define at least one persistence unit.

The extender will process each persistence.xml file it finds in the bundle and produce an instance of PersistenceUnitInfoService to correspond to each persistence unit. It will then register the instance in the service registry.

The persistence unit bundle may be part of a client application bundle (a caller of JPA), or it may be a separate bundle from the calling client, but if it is a separate bundle then it is assumed that the client will have classloading access either through OSGi package imports or as required bundles.

There may also be fragments attached to the bundle as long as they are attached before the bundle is resolved.

5.3.2 Persistence Unit Metadata

The PersistenceUnitInfoService interface provides an abstraction layer over the metadata contained in the persistence.xml file, as well as some additional information that can obtained by virtue of the extender having detected the bundle. The extender knows the bundle specifics and can pass the relevant information on to containers or providers.

5.3.2.1 persistence.xml Metadata

The persistence.xml metadata is represented as a Map of properties in the PersistenceUnitInfoService class. Each XML element or attribute corresponds to a dedicated property name in the Map, and the extender must mine the file to create the associated property values. If no element or attribute was present in the persistence.xml file there will be no entry in the Map for that property. Multiple XML elements are represented as properties with List values.

Property Name in Map	persistence.xml	Value Type
`javax.persistence.schemaVersion`	<persistence version=...> attribute	String
`javax.persistence.unitName`	<persistence-unit name=...> attribute	String

javax.persistence.transactionType	<persistence-unit transactionType=...> attribute	String
javax.persistence.provider	<provider> element	String
javax.persistence.jtaDataSource	<jta-data-source> element	String
javax.persistence.nonJtaDataSource	<non-jta-data-source> element	String
javax.persistence.mappingFiles	<mapping-file> elements	List<String>
javax.persistence.jarFiles	<jar-file> elements	List<URL>
javax.persistence.managedClasses	<class> elements	List<String>
javax.persistence.excludeUnlistedClasses	<exclude-unlisted-classes> element	String
javax.persistence.sharedCacheMode	<shared-cache-mode> element	String
javax.persistence.validationMode	<validation-mode> element	String
javax.persistence.properties	<property> elements	Properties

5.3.2.2 Additional Persistence Unit Metadata

Additional information must be constructed by the extender and filled in the PersistenceUnitInfoService instance.

Provider ServiceReference

The extender must obtain a ServiceReference to the provider. It chooses a provider by following an ordered set of steps. If a provider exists at any step then it is used.

1) Check if there is a provider specified in the persistence.xml file.

2) See if there is a default provider specified in the **defaultProvider** config admin property.

3) Discover available providers and pick one.

If a provider version was specified in (1) by the presence of the `osgi.jpa.provider.version` property, or in (2) by the presence of an additional **defaultProviderVersion** config admin property, then the version must also be considered in the provider resolving process, according to the standard version matching rules (i.e. the same matching rules that are used for Import-Package).

Persistence.xml Location

The extender must obtain and export a URL to the location of the persistence.xml file that contains the definition of the persistence unit. The URL may be in any format or protocol appropriate to the environment.

Persistence Unit Root

The extender must obtain and export a URL to the root of the jar in the bundle that contains the persistence unit. The URL may be in any format or protocol appropriate to the environment.

Persistence Unit Bundle

The extender must obtain and return the Bundle that contains the persistence unit.

Persistence Unit Classloader

The extender must create and export a classloader with visibility into the persistence unit bundle. This offers the provider a way to access the classes and resources there. A classloader proxy, or delegating classloader strategy, may be used, whereby calls to load a class or resource are forwarded to the bundle. Class instances and resources loaded by this classloader will be identical to those retrieved by calling Bundle.loadClass().and Bundle.getResource(). The way in which this is done may be specific to the underlying framework.

The proxy/delegator may also forward classloading calls to other classloaders, depending upon the needs and features of the implemention, particularly in a container that wants to allow classloading access to units of installation larger than a single bundle (e.g. a native “application”).

5.3.3 PersistenceUnitInfoService Service Properties

The PersistenceUnitInfoService instance should be registered in the service registry with the following service properties:

1) `osgi.jpa.persistence.unit.name`

   This property should have the persistence unit name as its value, and can be used to locate the persistence info for a given named persistence unit. The key constant is defined as `PersistenceUnitInfoService.PERSISTENCE_UNIT_NAME`.

2) `osgi.jpa.persistence.bundle.name`

   This property should have as its value the symbolic name of the bundle containing the persistence unit. The name can be obtained from the Symbolic-Name header in the manifest. The key constant is defined as PersistenceUnitInfoService.`PERSISTENCE_BUNDLE_SYMBOLIC_NAME`.

3) `osgi.jpa.persistence.bundle.version`

   This property should have as its value the version string of the bundle containing the persistence unit. The version string can be obtained from the Bundle-Version header in the manifest. The key constant is defined as PersistenceUnitInfoService.`PERSISTENCE_BUNDLE_VERSION`.

5.4 EntityManagerFactory Registration

In a non-managed environment a provider has the option to register a newly created EntityManagerFactory instance in the Service Registry. The boolean setting of the **registerEMF** config admin property will determine whether or not the provider should register the factory before it returns it to the caller. When registered, the factory should be accompanied with the `PersistenceUnitInfoService.PERSISTENCE_UNIT_NAME`, `PersistenceUnitInfoService.PERSISTENCE_BUNDLE_SYMBOLIC_NAME`, and `PersistenceUnitInfoService.PERSISTENCE_BUNDLE_VERSION` service properties.

Containers do not register EntityManagerFactory instances in the service registry. EntityManagerFactory or EntityManager instances may be injected into the injectable container components using standard JPA annotations.

Note: Registered factories can only be counted on to be used with the properties that were applied when it was initially created. Looking up an existing EntityManagerFactory in the service registry will normally not allow any configuration changes.

5.5 Weaving

There is no mechanism standardized within JPA for defining how to do weaving in the non-container case. Neither is there a standard OSGi mechanism to weave classes in an OSGi context. Some frameworks provide system level interception points that can be installed at framework startup to allow a weaver the option of being invoked whenever a class is loaded, but it is neither standard nor widely available. Providers must make use of framework-specific hooks.

Some possibilities that have been discussed and tried are:

1. Have a weaving service that uses a whiteboard pattern where providers that want to weave would register themselves as weaving services and the framework weaving hook would invoke all of them in turn as appropriate to their properties.

2. Use the Java SE 5 Instrumentation JVM hook, which provides a classloader interception point that precedes the application loading the class. This can be used by providers to hook in transformers to perform dynamic weaving for all classes.

In the managed environment the Container will need to have enough control over classloading in order to offer a javax.persistence.spi.ClassTransformer object to the provider through the PersistenceUnitInfo SPI.

5.5.1 Introduction of Dependencies

To resolve the problem of new dependencies being introduced by weaving, the imports of the woven bundle must be updated. The extender, when it detects a persistence bundle, can modify it such that it imports the necessary provider-specific packages. This could be performed through manifest-rewriting, attaching bundle fragments, or through some other mechanism.

The extender detects persistence units as it listens to INSTALLED bundle events. These events can be used to modify the bundle manifest before it is resolved, adding package imports to the persistence bundle such that any provider-specific code can be loaded once managed classes have been enhanced. The typical flow would be the following:

- An INSTALLED event is received for the persistence unit bundle.
- The extender identifies the persistence unit bundle by locating a META-INF/persistence.xml
- The extender parses the persistence.xml and then goes through the process of choosing a suitable persistence provider
- The extender uses the ServiceReference for the provider to locate the provider bundle in the OSGi framework
- The extender modifies the package imports in the persistence unit bundle to match the exports of the provider bundle.

5.6 Temporary Classloader

The provider may require a way to introspect the domain classes, as described in 2.2.2, option (2), without causing them to be loaded by their respective classloaders. The only way that it could do that is to load them in a different loader with the same classpath. This is called a temporary classloader, and the container is required to pass one to the provider in a managed environment. In a non-managed environment the provider may need to create one itself, but is not required to.

The classloader returned by PersistenceUnitInfo.getTempClassLoader should not cause classes to be loaded by the persistence unit bundle classloader. Class instances loaded by this classloader will not be visible to the bundle classloader. Calls to loadClass and getResource made on this classloader should produce the same result as if they were called on the Bundle interface for the persistence bundle.

One way of creating a temporary classloader with the same classpath as the “real” persistence unit classloader is to proxy the classloader with an additional loader that translates a loadClass call to a loadResource call, loading the class in as bytes and then actually defining it in the proxy loader. This averts it from being defined in the real domain class loader, causing it to instead be defined in the disposable proxy temporary loader.

5.7 JDBC Data Access

In a non-managed environment JPA applications specify the data source configuration using JDBC-style properties. These properties are used by the provider to look up and connect to the given JDBC driver, obtain connections from it, and use the connections to perform reads and writes to the database.

Access to the JDBC driver is made possible through the RFC 122 Database Access specification. A JDBC driver exports itself as a data source factory service in the service registry, and providers (or any other users of the data source) look up the factory and create data sources from it.

A sample snippet shows the properties that might be present in a persistence.xml file.

```
<properties>
    <property name="javax.persistence.jdbc.driver"
              value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:derby://localhost:1527/userDB"/>
    <property name="javax.persistence.jdbc.user" value="user"/>
    <property name="javax.persistence.jdbc.password" value="password"/>
</properties>
```

A provider, when a persistence unit is being processed, reads the properties and looks up the JDBC data source factory service. It filters by the driver class name property that is supplied in the persistence.xml file. If it can obtain the data source factory then it passes in the relevant URL and user information to obtain a data source. It may retain that data source to acquire connections throughout the life cycle of an entity manager factory.

```
DataSource ds;
ServiceReference[] refs =
        bundle.getServiceReferences("org.osgi.service.jdbc.DataSourceFactory");

for ( ServiceReference ref : refs ) {

  if (ref.getProperty(DataSourceFactory.JDBC_DRIVER)
            .equals("org.apache.derby.jdbc.ClientDriver") {

    Properties props = new Properties();
    props.put( DataSourceFactory.JDBC_URL, "jdbc:derby://localhost:1527/userDB" );
    props.put( DataSourceFactory.JDBC_USER, "user" );
```

```
    props.put( DataSourceFactory.JDBC_PASSWORD, "password" );
    DataSourceFactory dsf = ( DataSourceFactory )
            bundle.getBundleContext().getService( ref );
    ds = dsf.getDataSource( props );
    break;
  }
}
```

In a managed environment Containers may choose to support any number of ways for an application to specify a data source, but most container-based applications use the `jta-data-source` and `non-jta-data-source` elements in the persistence.xml file. The container may use the JNDI names listed in the elements in the persistence.xml file and look up the data sources internally, or in JNDI if it has implemented the transactional data source layer in JNDI.

5.8 Transactions

Non-managed environments tend to use local EntityTransaction transactions, meaning that the transaction is local to the entity manager and is not integrated with any other transaction. The EntityManager has full control over the transaction and manages the transactional resources through JDBC.

A managed EntityManager is expected to be integrated and synchronized with the active JTA transaction. The provider may be required to synchronize with the transaction either automatically by virtue of it being created within a transactional context, or through a joinTransaction() call made by the container. In order to determine the transaction status the provider assumes that a TransactionSynchronizationRegistry is available in the Service Registry or in JNDI, as specified by RFC 98. The provider looks up the transaction artifact in a managed environment and may retain a reference to the TransactionSynchronizationRegistry to query for transaction status when necessary.

5.9 Summary of Responsibilities

5.9.1 Extender Responsibilities

1. Detect Provider bundles
2. Register providers as services using the provider name and version service properties
3. Detect persistence unit bundles based on the presence of persistence.xml
4. Read all of the persistence unit metadata from the persistence.xml file
5. Associate a provider with each persistence unit (using the name and version resolving)
6. Add exports from associated provider to persistence unit bundle imports
7. Create PersistenceUnitInfoService from persistence unit metadata and additional context info
8. Register persistence units as services with persistence unit name and version as service properties

5.9.2 Provider Responsibilities

1. When createEntityManagerFactory() is invoked, look up the PersistenceUnitInfoService for the named persistence unit and see if the provider is set to the calling provider

2. Use the information from the PersistenceUnitInfoService to access the classes and resources to create an EntityManagerFactory

3. Use the JDBC service to access the driver (or JTA data source in a container) and obtain a data source from which it can obtain connections.

5.9.3 Container Responsibilities

1. Look up all PersistenceUnitInfoService services and determine which ones are container-managed persistence units

2. Create a PersistenceUnitInfo for each container-managed persistence unit and call the createContainerEntityManagerFactory method on the associated provider

5.9.4 Persistence API Bundle Responsibilities

1. Implement PersistenceProviderResolver to look in service registry for providers

2. Set resolver in resolver holder

5.10 Limitations

The following limitations may exist in compliant implementations of this specification:

1. Support for the configuration outlined in 2.4.6 is not required.

2. All defined persistence units in the same bundle are assumed to be provided by the same provider.

3. Once a persistence bundle has been associated with a provider the provider cannot be changed without re-resolving the persistence bundle and potentially re-writing the package imports. Any and all clients using the persistence bundle would also need to be stopped and re-resolved after the persistence bundle had been “re-wired” to a different provider.

4. A persistence bundle may only be wired to a single provider that supports weaving. If a persistence bundle defines multiple persistence units they must all be associated with the same provider.

5. Persistence bundles may need to include all of the classes and resources. If the classes or resources are in a separate bundle from the persistence bundle there may be no way of knowing which bundle they are in and it would be difficult to rewrite the manifest of that bundle.

6 Security Considerations

The JPA provider may need permissions in order to perform weaving or reflection, depending upon the configuration or even provider implementation choices. These security considerations are no different than those that exist outside of OSGi, and do not need to be given special consideration.

7 Considered Alternatives

Much of the bundle snooping was originally implemented by the provider, requiring each provider to duplicate much of the same logic and complicating the process of running a provider in OSGi. The extender class is the logical place to implement it, with all providers being able to benefit from the shared code.

One way to export the persistence information was to let the extender go all the way to creating the entity manager factory and registering it in the Service Registry. The container would then look up the EMF and use it to create entity managers and inject them into components, etc. The decision was that containers might want to use their own techniques and have their own implementations that used vendor-specific hooks and features.

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

8.2 Author's Address

Name	Mike Keith
Company	Oracle Corporation
Address	45 O'Connor Street, Ottawa, ON, Canada K1P 1A4
Voice	+1 613 288 4625
e-mail	michael.keith@oracle.com

Name	Timothy Ward
Company	IBM Corporation
Address	IBM United Kingdom Limited, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom.
Voice	+44-1962-816874
e-mail	timothy.ward@uk.ibm.com

Name	Graham Charters
Company	IBM Corporation
Address	IBM United Kingdom Limited, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom.
Voice	+44-1962-816527
e-mail	charters@uk.ibm.com

9 Appendix A - PersistenceUnitInfoService

```
package org.osgi.jpa;

import java.net.URL;
import java.util.Map;

import org.osgi.framework.Bundle;
import org.osgi.framework.ServiceReference;

/**
 * This interface exposes the information found or created by RFC 143 when
 * discovering persistence units. A service will be registered in the service
 * registry for each persistence unit discovered.
 */
public interface PersistenceUnitInfoService {

	/* ================================================================
	 * Constants section
	 * ================================================================
	 */

	/* ================================================================================
	 * The following constants are used as keys in the persistenceXmlMetadata
	 * Map and represent elements or attributes of a persistence unit in a
```

```
     * persistence.xml file. If the elements were not specified in the
     * persistence.xml file then the corresponding entries will not be present
     * in the persistenceXmlMetadata Map.
     * ========================================================================
     */

    /**
     * The key used to store the schema version of the persistence.xml file.
     * This entry should always be present and will always contain a value.
     *
     *   persistence.xml - <persistence version=...> attribute
     *   value type - String
     */
    public static final String SCHEMA_VERSION =
                                  "javax.persistence.schemaVersion";

    /**
     * The key used to store the persistence unit name.
     * This entry should always be present and will always contain a value.
     *
     *   persistence.xml - <persistence-unit name=...> attribute
     *   value type - String
     */
    public static final String UNIT_NAME = "javax.persistence.unitName";

    /**
     * The key used to store the transaction type.
     *
     *   persistence.xml - <persistence-unit transaction-type=...> attribute
     *   value type - String
     */
    public static final String TRANSACTION_TYPE =
                                "javax.persistence.transactionType";

    /**
     * The key used to store the provider class name.
     *
     *   persistence.xml - <provider> element
     *   value type - String
     */
    public static final String PROVIDER_CLASSNAME = "javax.persistence.provider";

    /**
     * The key used to store the JTA data source.
     *
     *   persistence.xml - <jta-data-source> element
     *   value type - String
     */
    public static final String JTA_DATASOURCE = "javax.persistence.jtaDataSource";

    /**
     * The key used to store the non-JTA data source.
     *
     *   persistence.xml - <non-jta-data-source> element
     *   value type - String
```

```
     */
    public static final String NON_JTA_DATASOURCE =
                                "javax.persistence.nonJtaDataSource";

    /**
     * The key used to store the List of mapping file names.
     *
     *   persistence.xml - <mapping-file> elements
     *   value type - List<String>
     */
    public static final String MAPPING_FILES = "javax.persistence.mappingFiles";

    /**
     * The key used to store the List of jar file locations.
     *
     *   persistence.xml - <jar-file> elements
     *   value type - List<URL>
     */
    public static final String JAR_FILES = "javax.persistence.jarFiles";

    /**
     * The key used to store the List of managed class names.
     *
     *   persistence.xml - <class> elements
     *   value type - List<String>
     */
    public static final String MANAGED_CLASSES  =
                                "javax.persistence.managedClasses";

    /**
     * The key used to store whether unlisted classes should be excluded.
     *
     *   persistence.xml - <exclude-unlisted-classes> element
     *   value type - Boolean
     */
    public static final String EXCLUDE_UNLISTED_CLASSES =
                                "javax.persistence.excludeUnlistedClasses";

    /**
     * The key used to store the shared cache mode.
     *
     *   persistence.xml - <shared-cache-mode> element
     *   value type - String
     */
    public static final String SHARED_CACHE_MODE  =
                                "javax.persistence.sharedCacheMode";

    /**
     * The key used to store the validation mode.
     *
     *   persistence.xml - <validation-mode> element
     *   value type - String
     */
    public static final String VALIDATION_MODE  =
                                "javax.persistence.validationMode";
```

```
/**
 * The key used to store the persistence unit properties.
 *
 *   persistence.xml - <property> elements
 *   value type - Properties
 */
public static final String PROPERTIES = "javax.persistence.properties";

/* ================================================================
 * The following constants are used as service property keys when
 * registering this object as a service in the service registry.
 * ================================================================
 */

/**
 * The service property key used to store the persistence unit name.
 */
public static final String PERSISTENCE_UNIT_NAME =
                                "osgi.jpa.persistence.unit.name";

/**
 * The service property key used to store the bundle symbolic name.
 */
public static final String PERSISTENCE_BUNDLE_SYMBOLIC_NAME =
                                "osgi.jpa.persistence.bundle.name";

/**
 * The service property key used to store the bundle version.
 */
public static final String PERSISTENCE_BUNDLE_VERSION =
                                "osgi.jpa.persistence.bundle.version";

/* ================================================================
 * The following properties are defined here for convenience.
 * ================================================================
 */

/**
 * The key used to register the JPA provider version service property.
 * It may also be used as a persistence.xml property key to specify a
 * provider version range for the persistence unit.
 */
public static final String JPA_PROVIDER_VERSION = "osgi.jpa.provider.version";

/* ================================================================
 * Methods section
 * ================================================================
 */

/**
 * Expose the metadata for a particular persistence unit. The metadata is
 * obtained by parsing the elements and attributes for the persistence unit
 * defined in a persistence.xml file. Entries in the Map will have keys and
 * values as defined by the constants above.  Changes to the returned Map
 * will not be reflected in the original Map.
```

```
	 *
	 * @return A copy of the metadata Map
	 */
	public Map<String, Object> getPersistenceXmlMetadata();

	/**
	 * A provider has been associated with this persistence unit, either because
	 * the provider was listed in the persistence.xml file, or was defaulted.
	 *
	 * @return A ServiceReference to the provider that is expected to service
	 *         this persistence unit
	 */
	public ServiceReference getProviderReference();

	/**
	 * Return a URL that may use an OSGi-specific protocol. In order to pass the
	 * URL in a PersistenceUnitInfo, in a format the provider can use to stream
	 * over, the container may need to convert this URL to a file-based one.
	 *
	 * @return A URL to the location of the persistence.xml file that defined this
	 * persistence unit
	 */
	public URL getPersistenceXmlLocation();

	/**
	 * Return a URL that may use an OSGi-specific protocol. In order to pass the
	 * URL in a PersistenceUnitInfo, in a format the provider can use to stream
	 * over, the container may need to convert this URL to a file-based one.
	 *
	 * @return A URL to the location of the persistence unit root
	 */
	public URL getPersistenceUnitRoot();

	/**
	 * Provide a reference to the bundle in which the persistence unit was
	 * detected. The bundle may be a client bundle, or it may be a standalone
	 * persistence unit bundle.
	 *
	 * @return The bundle in which the persistence.xml file was detected
	 */
	public Bundle getDefiningBundle();

	/**
	 * A classloader with visibility to the classes and resources in the bundle
	 * in which the persistence unit was detected.
	 *
	 * @return Classloader that can access persistence unit classes and resources
	 */
	public ClassLoader getClassLoader();

}
```

9.1 End of Document