RFP 115 - JPA Integration

Draft

11 Pages

Abstract

This document lays out requirements to support usage of the Java Persistence API inside an OSGi Framework.

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 5.1.

```
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	Oct 31 2008	Initial draft. Mike Keith, Oracle Corporation michael.keith@oracle.com
0.8	Jan 9 2009	Mike Keith, Oracle Corporation michael.keith@oracle.com
1.0	Feb 9 2009	Draft for vote. Mike Keith, Oracle Corporation michael.keith@oracle.com

1 Introduction

The Java Persistence API is integated with the Java EE Container but can also run in a Java SE environment. Because of its standard API, O-R mapping metadata, and configuration metadata it is becoming the standard persistence framework for mapping Java objects to relational databases.

This document describes the requirements that must be met in order for JPA implementations (persistence providers) to be obtained and used by standalone and enterprise applications in an OSGi environment. It may also include expectations on clients, in order for them to use JPA in an OSGi platform.

The ideal way to meet all the requirements described in this document would require that we not only add features to the OSGi Core Framework and/or Compendium Services, but that we make changes to JPA via the Java Community Process (JCP). With different release cycles and priorities, however, aligning these two may not be achieveable in the short term, thus the goal is to solve as many use cases as we can within the current OSGi context.

2 Application Domain

JPA is a critical layer in any enterprise Java application since it represents a link to the outside data world, without having to embed database-specific code in the application. It is used to read and write to the relational database through deployed enterprise data sources, a commonly used feature of Java Enterprise Edition (Java EE) environments. It is also used by Java SE applications to directly access databases through JDBC drivers, or to test enterprise applications outside of the usual EE container.

Virtually any application can use JPA to meet their ORM persistence needs, thus the application domain is almost as wide and deep as the application space is itself. Certainly the traditional enterprise applications like web, business and IT applications commonly use JPA and similar persistence solutions to connect to RDB technology.

JPA runs in two different modes, depending upon the environment that it is being used in. When running in Java SE, or a non-container environment, the bootstrap Persistence API class is used with local transactions. When used in a Java EE container, or any other compliant JPA container host, the container provides persistence provider detection and loading, as well as injection services of certain useful JPA resources. Applications that rely on injection will expect it to be provided wherever their application is running, hence an OSGi framework must continue to support applications that make use of injected “container-managed entity managers”. This RFP pertains to and includes applications that use both forms of the API when using JPA.

2.1 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 SPI, invokes JPA providers, and injects EntityManager and EntityManagerFactory instances into client classes

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

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

One of the core architectural benefits of JPA is the ability for a JPA client to be isolated from the particular implementation, or JPA provider, that is offering the JPA service. Clients may be compiled with only the JPA interface JAR on the classpath, but may call into the API at runtime. If there are no implementation-specific dependencies in the calling code, and the persistence unit does not include any such dependencies, then any JPA provider on the classpath can be used to back the persistence unit and provide the required persistence functionality. This ability to "plug in" a persistence provider, although similar in spirit to the goals of OSGi, ironically poses a problem in the OSGi integration because the module decoupling that allows the separation of a JPA client from a JPA provider is actually resolved using a shared classpath mechanism that is not a feature of OSGi.

Although JPA does not require that providers offer transparent lazy loading of many-to-one and one-to-one relationships, it is supported by most major JPA providers and is a common feature that users expect. In order to support it the provider must have access to either a dynamic code generation service to generate a subclass or proxy, or a byte code transformation service in connection with the container classloader or via an Instrumentation hook provided at JVM startup.

3.1 Java SE Perspective

When using the bootstrap Persistence API class, the JPA Provider is expected to read the JPA resource files, possibly weave the managed classes, and maintain metadata for specific classes. The classloader, therefore, plays a substantial role in how the provider manages JPA persistence units and the associated resources and classes, with the context classloader being the loader that it traditionally uses to do it.

The first problem is that of provider detection. In the event that no provider is specified by the application, an available provider must be sought out to provide support for JPA. In order to find all available providers, the Persistence bootstrap class currently examines the classpath for service provider description file resources that a provider exports to indicate its support for the JPA SPI. In the presence of different vendor bundles being loaded, a Persistence class would not typically have access to the resources in those loaded bundles.

The next phase involves the provider being able to find and read the persistence.xml file that is part of the client application. This file contains the relevant persistence unit configuration for the provider to be able to initialize its internal structures. Since the provider has no real dependencies on the client application, but is rather being invoked indirectly, any persistence.xml files bundled in the application would not be visible to the provider in OSGi.

As part of the processing of the persistence unit the provider will typically need to examine the domain classes (that are part of the application) for annotation metadata. Some providers use byte code weaving to modify the managed classes based upon the configuration. In normal Java SE environments the Instrumentation JVM hook is used to intercept classloading of domain classes. This offers the provider an opportunity to weave a managed class before it is loaded by the application. There is no such facility defined in OSGi, particularly across bundles.

3.2 Java EE and Host Container Applications

When used in a Java EE container, or any other compliant JPA container host, the container is expected to read and process specific JPA resources, load one or more JPA providers, provide a byte code transformation service to the provider, control the life cycle of container-managed entity managers, associate entity managers with transaction contexts, inject JPA resources into specific client classes, and more. In the container scenario the container is typically in control of the environment and able to offer the kinds of services that a provider requires; however, in OSGi the necessary control is not in the hands of any one particular service. It is up to the container

implementation to provide the container services according to normal Java EE injection practices and transaction integration.

The first step in a container-driven JPA deployment is for the container to find the persistence.xml resource in the deployment classpath. The file is parsed and processed, and an entity manager factory is created for each persistence unit. There may in fact be multiple persistence.xml files within any given deployment, and multiple persistence units within any given persistence.xml file. The container must process each and every one that is found.

Container deployments do not require the list of managed classes to be specified. When running in a container the provider is given a classloader that it can use to search the deployment unit to locate the entities, embeddables and mapped superclasses. The union of classes that are either annotated to be a managed class or included as a mapped class in an XML mapping file is the set of managed classes that is assumed by the provider to be persistable within a given persistence unit.

The container services that are needed by the provider are made available through an SPI data structure that is passed to the provider. This includes a current deployment unit classloader, a temporary classloader that allows temporary loading of classes (to be discarded when the classloader is discarded), a class transformer to weave classes at load time, and one or more data sources that are accessible to the container and made available to the provider. Resource and location URLs, and properties, are also passed to the provider to allow it to open and process XML mapping files.

Entity managers and factories are typically injected into container-managed components. Designed for Java EE environments these components are normally EJB's or servlets, but the container environment and component model that is in use may not be an EJB or Java EE environment, as long as it supports injection and life cycle management of injected resources.

Another of the JPA container responsibilities is to associate the life cycle of a transactional entity manager with a global transaction. In the case of a transactional component calling another transactional component then if the transaction is propagated across the call to the target component then an injected entity manager instance should also be propagated to the target component.

4 Use Cases

4.1 Java SE application independence from JPA Provider

A Java SE application is compiled with the JPA interface JAR on its classpath and does not need to specify a provider in the accompanying persistence unit definition in persistence.xml. At runtime it expects a provider to be found and used as the JPA provider.

The application is loaded in a single bundle that includes a META-INF/persistence.xml and all of its entities and domain classes. The API jar and the provider are each loaded as separate bundles.

The provider must be able to detect, access and load the META-INF/persistence.xml file from the client bundle and process the elements contained within it. The provider must also be able to load the entity classes, even though it has no prior knowledge of, or a priori dependencies on, those classes.

4.2 Container application independence from JPA Provider

Applications that operate within a container environment are similarly compiled against the JPA interfaces without any provider-specific classes being required. When no provider is specified in the persistence.xml file then at runtime the container will choose a default JPA provider.

The container must be able to detect, access and load the META-INF/persistence.xml file from the client bundle and process the elements contained within it. The container must also be able to load the provider classes, whether they are specified or defaulted, and pass the container-supplied SPI data structure and client-defined properties to the provider.

4.3 JPA Provider bundling JPA interfaces

Some JPA providers may combine the JPA interfaces with the implementation classes and create a single bundle. Applications should not need to be aware of the bundling specifics of the JPA provider as their dependencies are strictly interface-based.

4.4 Combining annotations metadata and mapping files

Mapping metadata may be specified in the form of annotations on the domain classes and/or XML mapping files. A default mapping file called orm.xml may be present, with additional mapping files possibly listed in the persistence.xml file.

Although mappings are most often expressed in annotation form, it is also a somewhat common practice to specify a specific portion of the metadata in an XML mapping file, or in some cases override the annotation metadata with an XML file containing metadata particular to a specific database or production environment.

4.5 Reliance on Provider Features

Every JPA provider offers features above and beyond the JPA specification, and a great many applications end up making use of at least one vendor-specific addition.

Vendor features come in different forms:

- persistence properties
- query hints
- vendor implementation classes
- vendor metadata and annotations

The persistence properties, and for the most part the query hints, have no effect on the OSGi provider configuration. However, domain classes associated with an application may contain code that references a vendor class, or may be decorated with vendor-specific annotations. These hard dependencies require that the application import vendor packages, and represent a coupling to a particular JPA provider, but should not cause any other undue problems.

4.6 Persistence archive

The ability to modularize and reuse a persistence unit is just as valuable for a persistence-based application as any other. In JPA it is possible to create a persistence archive, or a JAR file that contains the domain classes, optionally some mapping files, and a META-INF/persistence.xml file that specifies the persistence unit configuration details.

A persistence archive represents a reusable package that can be imported into any application. An application bundle can import a persistence archive bundle and use the domain classes as persistence components. A provider would need to be able to access and load those domain classes, even though they are not part of the application bundle that is importing JPA.

4.7 Multiple persistence units

When an application needs to connect to multiple data sources it typically creates a persistence unit for each data source. There is only necessarily one version of the domain classes (bundled in the application, or in a referenced JAR), but multiple mapping views from those domain classes to the different data sources. At least one of the persistence unit mapping groups is specified in a mapping file, although they are often both are in separate mapping files.

The application will make use of each of the persistence units for different data source access use cases, and hence may have multiple open entity managers active but configured for different target databases.

4.8 Changing Providers

Applications with no dependencies on any particular JPA provider may be flexible enough to switch providers. Unloading the provider bundle and loading a bundle from a different provider offers a convenient migration and testing path for applications to try out the performance and behavior of different providers.

4.9 Upgrading the Provider

The prototypical OSGi advantage of upgrading a bundle to a more recent version applies equally well to a provider bundle, as well as to the JPA spec bundle. At an appropriate time an existing provider version may be unloaded and a new version of the provider loaded in its place. Dependence on the provider should make no difference to this use case.

4.10 Support Different Implementation Strategies

The JPA specification goes out of its way to ensure that it does not force persistence vendors to implement the specification in a particular way or using a particular strategy or design. Some providers might want to use code generation while others use byte code weaving as a means to provide transparent entity persistence. The fact that different providers have chosen different strategies bestows different strengths and weaknesses to each, allowing an application more choice, and the ability to use the most appropriate one according to its access and usage patterns.

4.11 Simple Container Deployment Units

Deploying managed classes in a container typically involves enclosing them in a deployment unit or on the classpath of a deployment unit. There is not currently any recognized deployment unit in OSGi other than a bundle or some variation of extensions and/or fragments. Simple JAR deployments, similar to that of an EJB-JAR, should pose no problems to container-hosted scenarios.

4.12 Container Detection of Managed Classes

Entities, embeddables and mapped superclasses are not normally listed in <class> elements in the persistence.xml file. Instead, the container must be able to find/detect classes that are persistable by virtue of them being on the classspath and having an indicating annotation on them, or being listed/mapped in a default

META-INF/ORM.XML mapping file, or separate mapping file that was listed in the persistence.xml file. This provides a mechanism for being able to add new entities, refactor the entity packages, or remove entities without having to continually update the persistence.xml file and explicitly list the fully qualified class names.

4.13 Lazy Loading of Relationships

For almost any graph of entites the cost of loading the entire graph when only one or two levels is accessed is in the best case wasteful and an inefficient use of resources, and in the worst case prohibitively expensive and untenable. To prevent this situation from happening, relationships may be configured to be lazily loaded.

While a relationship specified as LAZY is only a hint to the provider and may not necessarily be loaded lazily, many vendors support the notion of lazy loading. Vendors that support lazily-loaded relationships will be able to function as they do in other environments, using proxy subclass code generation, byte code weaving, or whatever implementation strategy they have chosen to achieve it.

4.14 Application-managed Entity Manager

An entity manager factory may be injected as a dependency into a component, or obtained by calling Persistence.createEntityManagerFactory(), and once obtained may be shared across threads. It is also used to acquire an application-managed entity manager that can be passed around within the same thread. Application-managed entity managers allow extensive application control over the entity manager life cycle and may be used to retain entity managers across transactions.

4.15 Container-managed Entity Manager

An entity manager that is obtained directly from the container, either from a JNDI lookup or as a result of dependency injection, has a life cycle that is controlled by the container, and the application does not need to do anything other than use it within the scope for which it is defined (either transaction or extended scoped when in a stateful component). They are the most convenient type of entity manager and offer an application the most ease of use for an operation or series of operations that are of simple or intermediate complexity.

4.16 Using Local Transactions

When short, single-resource transactions to a database are being employed then JDBC level transactions are appropriate. A JPA EntityTransaction models this local transaction, and is scoped privately to the specific entity manager from which the transaction is demarcated. An application may choose to use an EntityTransaction on an entity manager that is configured to support them (but may not initiate EntityTransactions on entity managers that are configured for global transactions).

4.17 Global Transactions and Propagation

Components that support automatic transaction demarcation come with transactional guarantees according to declarative attributes that are specified for the component. When such components are used the container transaction is usually an industrial-strength transaction that provides support for multiple resources, 2-phase commit, and so forth. In present-day Java terms this equates to a JTA transaction that is global to the services running in a particular address space,

When configured to use JTA transactions, an entity manager participates in and is synchronized with the global JTA transaction. If transaction-scoped, the entity manager would be disposed of after the transaction has been completed, but in the case when a transaction is propagated during the course of the transaction demarcation rules, the managed entities within an entity manager is propagated to any entity manager in the component that is inheriting the transaction.

5 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.

5.1 References

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

[2]. Java Persistence API, May 2006, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html .

5.2 Author's Address

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

5.3 End of Document