RFP-170 Managed JPA

Final

11 Pages

Abstract

The OSGi Alliance Enterprise Expert Group has developed adaptations of the JDBC, JTA, and JPA Java EE specifications. These resulted in the not-managed model, that is they provide factory services (DataSourceFactory and EntityManagerFactoryBuilder) and a JTA Transaction service. However, the native OSGi model is to be able to use configured instance services so they can directly be injected through DS. This RFP analyzes the current state of the art and seeks a proposal for a persistence specification that fully leverages the OSGi service model and Java 8.

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 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	*02/12/14*	*Initial* *Peter.Kriens@aQute.biz*
0.1	*21/01/15*	*Update to include more information about transaction integration, connection pooling and security of credentials.*
0.2	*06/05/15*	*David Bosschaert – Updates as discussed at the Cologne F2F.*

1 Introduction

The EEG has developed adaptations of the Data Source and JPA Java EE specifications. These resulted in the not-managed model, that is they provided factory services (Data Source Factory and Entity Manager Factory Builder) and a JTA Transaction service. However, the true OSGi model is to be able to use configured instance services so they can directly be injected through DS. This RFP analyzes the current state of the art and seeks a proposal for a persistence specification that leverages the OSGi service model and Java 8.

This RFP originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that need to be solved.

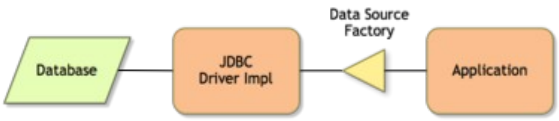
2 Application Domain

Persistence is one of the most important aspects of modern applications. The current mainstream standard for Java is the Java Persistence Architecture (JPA), an Object Relational Mapping (ORM) framework that allows developers to work with objects and builds the corresponding SQL to persist those objects in a relational database.

JPA collaborates with the Java DataBase Connectivity (JDBC) and the Java Transaction Architecture (JTA) specifications. In the first OSGi Enterprise Release these Java EE specifications were adapted to OSGi to make them service based. Together these are called the OSGi *persistence services*. The OSGi persistence services are very much *factory* oriented instead of *instance* based. Instance based services are ready to use, they are usually configured by Configuration Admin, and have their dependencies resolved.

2.1.1 Data Source Factory

The JDBC Data Source Factory specification describes how a *database driver* can register a Data Source Factory service.



The Data Source Factory Service provides a number of methods to create a *Data Source*. A Data Source configures the underlying database and then provides a way to get the database *connections*. Since connections are expensive objects, implementations of a Data Source are often required to *pool* these connections. Pooling reuses connections for other requests after they are closed. Many libraries have been developed to optimize this pooling; these libraries often act as an intermediate between the actual Data Source or Driver and the application. They often use dynamically generated proxies since the JDBC API has gone through several non-backward compatible changes. This is so common that the API supports methods to unwrap these proxies.

There is also an XA Data Source, which provides access to the XA 'resource' protocol. The XA protocol is used by databases and other transaction aware resources to participate in transactions.

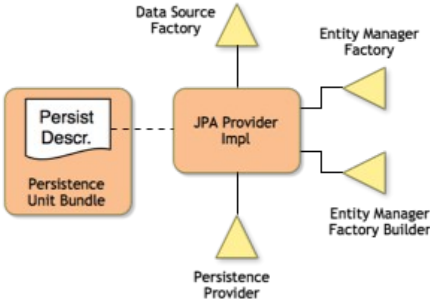
Since the underlying connections can be pooled, it is crucial that operations are properly *scoped*. That is, any obtained connections must be closed to allow them to returned to the pool. Connection Pooling is typically achieved using a library which “wraps” the DataSource. BoneCP, Apache DBCP and C3P0 are examples of libraries which offer this functionality.

In the OSGi enRoute project the Data Source Factory model was extended with a component with a well known PID and set of configuration properties to automatically register Data Source services.

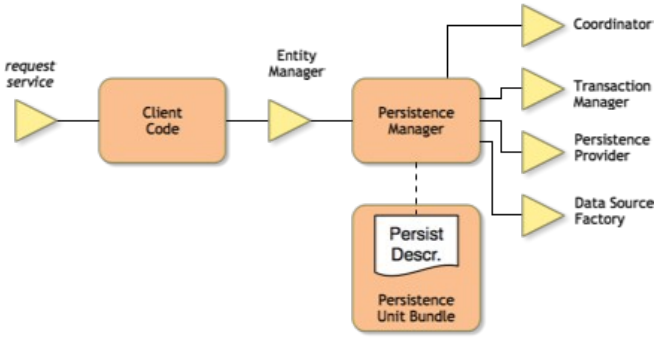
2.2 JPA

The JPA specification defines how a JPA *provider* can discover a *persistence descriptor* in a bundle. After this discovery, the provider registers an Entity Manager Factory when the persistence descriptor has sufficient information and can be associated with a Data Source Factory. It also registers a Entity Manager Factory Builder that can be used by the application to provide additional properties for configuration and that can create an Entity

Manager Factory. Applications that created an Entity Manager from this factory whenever they had to execute a request. Entity Managers are not thread safe.

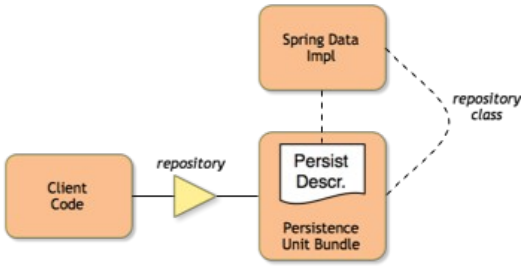


In OSGi enRoute, a component was defined with a well known PID and properties schema that registered an Entity Manager if an appropriate Data Source was available. This required the Entity Manager to proxy an actual Entity Manager since now the life cycle could be managed per thread. All requests would start a Transaction for the request thread. This allowed the Entity Manager proxy to detect the first request and it would then join the Transaction Synchronization Registry. At the end of the transaction, the resources used by the Entity Manager liked pooled connections could then be cleaned up automatically. A similar model is used in Apache Aries (### yes?).



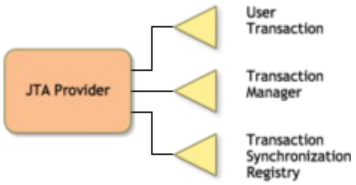
2.3 Spring Data

Christian Baranowski [4]. was inspired by OSGi enRoute and Spring Data. Spring Data provides a model where the methods on an interface are used to specify the query. That is, `findBlogByTitle("hello")` would translate to a request to the database to retrieve the blogs that match the title “hello”. He provides an implementation that gets the class name for the repository, which must extend a provided base class, and then registers the repository service for that persistent unit.



2.4 Transactions

The OSGi JTA specification only provides access to the different Transaction objects via services. It does not provide any other features than that are available from the existing JTA specification.



2.4.1 Client-side Transaction management

A Transaction is started by a client by beginning a transaction. It then executes the request code inside a block. If the requests is successful, the transaction is committed, otherwise it is rolled back. In general the catch block and finally block are used to ensure proper termination of the transaction. For example:

```
Transaction transaction = tm.getTransaction();
try {
   … do work
   transaction.commit();
} catch( Throwable t) {
   transaction.rollback();
   throw t;
}
```

In general, applications must minimize any code inside a transaction since transactions lock database tables. In applications build from different parts, this creates the Transaction Composability problem, see [3]. When a method gets called it can be called inside a transaction or outside a transaction. However, it can require that no transaction is active, that a transaction should be active, or that it requires a new transaction. Handling this inline increases the boiler plate code significantly. For this reason, Spring provides transaction annotations.

```
@Transactonal(readOnly = false, propagation = Propagation.REQUIRES_NEW)
public void doWork() {...}
```

These annotations require a proxy that sets up a transaction before the method is executed. This implies that the method cannot call methods on the `this` object since this bypasses the proxy.

In containers like Java EE the transaction manager is often not directly used. The container provides an Entity Manager that is already associated with a transaction manager. Since the container knows when a request is finished, it can perform any required cleanup. This called *managed transactions.*

2.4.2 Infrastructure integration with Transactions

For work to be included in a transaction It is necessary, but not sufficient, for a client to run code within a transaction. The other critical point is that any resource access that occurs within the transaction must be registered with the transaction manager. Typically this work is not performed by the client, but by a middleware component. In a Java EE application server this role is usually performed a JCA Resource Adapter.

The integration point is responsible for registering the connection with the transaction. If a two-phase commit (XA) transaction is needed then the XAResource associated with the XAConnection must be registered with the Transaction. This enables the Transaction manager to log the details of the resource (to enable recovery) and to prepare/commit/rollback any work. If no XA transaction is required then the resource adapter must register a synchronization with the transaction. This synchronization is responsible for rolling back or committing the work performed with that connection.

2.5 Versions and Migration

Code has an actual dependency on the layout of the database but this tight version dependency is rarely managed. New bundles often require a *migration* of the database but do not have standard support. New columns and tables must be added and sometimes data must be converted. This is often a manual error prone operation. An interesting development is Liquibase [6]. and Flyaway [7]. These projects allow refactoring of databases.

2.6 Terminology + Abbreviations

3 Problem Description

The Java EE persistence model with its statics and factories is not natural for OSGi and therefore offers a number of challenges to use. Additionally, the primary providers of JPA Hibernate, OpenJPA, and Eclipselink have not adopted OSGi which makes it hard to find a full implementation. The Apache Aries project provides most parts but the Gemini project has all but died. However, even with a full implementation the model is not trivially to use since it still requires the client code to handle a lot of configuration details.

Overall, JPA persistence has its problems in Java EE due to portability problems but in OSGi using JPA is really hard and clearly does not provide the plug and play as well as the collaborative component model that OSGi promotes. The current specifications are not very well matched to the OSGi service model that is configured instance based and not factory based (let alone statics).

Additionally, there are a number of promising developments on the horizon.

- Java 8 lambdas will make it easier to use transactions in a way that is as easy as annotations without the corresponding drawbacks
- The Spring Data JPA Repository looks very interesting for simple database models. Though large enterprise applications might have no use for this, it would lower the threshold for OSGi if it was easier to get started with small models. Obviously there should be a migration path to go to the Entity Manager and Data Source.
- Migration of databases and dependency management on the installed db version is becoming increasingly important. Obviously, the OSGi require-capability and extender model make excellent mechanisms to provide this kind of support.

This RFP therefore seeks a comprehensive service based model for persistence in OSGi leveraging Java 8 features.

4 Use Cases

```
## have not verified any of the following code
```

4.1 Blog

AI Bundle is making a demo and uses a simple Blog application. AI defines his BlogPost classes and defines a `persistence.xml` file that are stored in a bundle.

```
@Entity
public class BlogPost {
  @Id
  @GeneratedValue
  public Long id;
  public String text;
  public String author;
  public Long created;
}
```

Additionally, he creates the repository interface:

```
public class BlogRepository extends JPARepository<BlogPost,Long> {
  List<BlogPost> findByTitleContaining(String phrase);
}

manifest:
Bundle-JPA-Repository: com.example.blog.BlogRepository
```

He now writes his code.

```
@Component
public class Blogger {

  public List<Blog> getBlogs() {
     return br.findAll();
  }

  @Reference
  void setBlogRepository( BlogRepository<BlogPost,Long> br) {
    this.br = br;
  }
}
```

4.2 Blog and Transactions

This use case extends 4.1. AI needs to know the latest blog. He needs to rename one of the authors and he thinks this needs to be done in a transaction:

```
public int renameAuthor(String from, String to) {
  return br.requireTransaction( () → {
    br.findByAuthor(from).forEach( (a) → a.author=to );
  });
}
```

4.3 Blog and Entity Manager

This use case extends 4.1. Al found out that it is actually cheaper to do the update in the database directly. To do this, he needs the Entity Manager and perform a SQL statement.

```
public int renameAuthor(String from, String to) {
  EntityManager em= br.getEntityManager();
  Query q = em.createNativeQuery(
     "update blogpost set author=\":to\" where author=\":from\"")
  q.setParameter("to", to);
  q.setParameter("from", from);

  return br.requireTransaction( () → q.executeUpdate());
}
```

5 Requirements

5.1 General

- G0010– The service solutions must be able to work with JDBC 4 and JPA 2.1
- G0020 – Provide a simplified way to handle the transaction composition that does not require boiler plate code nor suffers from the problem that methods on the 'this' pointer are different..
- G0030 – The solution must not require that configured Data Access resources are publicly available. This prevents other bundles having access to database credentials.

5.2 Data Source Service

- D0010 – Provide a configuration model for a Data Source service
- D0014 – Must be able to transparently register connections with an ongoing transaction.
- D0017 – Must be able to support both two-phase (XA) resources, and non XA resources
- D0020 – Must be able to transparently handle connection pooling of “plain” DataSources
- D0030 – Must support all OSGi JDBC Factory defined possibilities and properties
-

5.3 Entity Manager Service

- E0010 – Provide a configuration model for an Entity Manager service

- E0020 – Must be able to use the Data Source service from 5.2
- E0030 – Must be able to have managed transactions. That is, the client uses an injected Entit Manager.
-
- E0050 – Must provide access to the underlying Data Source

5.4 Database Versioning

- V0010 – Provide a require-capability model for handling the version of the database
- V0020 – Provide a model so that bundles can be used to migrate and rollback a database.

6 Document Support

6.1 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

[3]. http://blog.osgi.org/2013/11/the-transaction-composability-problem.html
[4]. http://www.slideshare.net/tux2323/osgi-and-spring-data-for-simple-web-application-development
[5]. http://projects.spring.io/spring-data/
[6]. http://www.liquibase.org/
[7]. http://flywaydb.org/

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

Name	Tim Ward
Company	Paremus
Address	
Voice	
e-mail	tim.ward@paremus.com

6.3 End of Document