Beyond Java ORM with Versant JPA (Part 1)

German Viscuso
Developer Relations Manager
Versant Corporation
March 2012

Back in 2004 Sun Microsystems was struggling with the shortcomings of its Enterprise Java Beans (EJB) 2.0 and EJB 2.1 specifications. It was clear at the time that simplification of the architecture, standardization and persistence enhancements were needed to ease the lives of Java developers.

Thus the Java Persistence API 1.0 specification was born in 2006 as part of EJB 3.0 (JSR-220) with further enhancements in 2009: JPA 2.0 (JSR-317). It was a welcome standard that included support for many of the features that EJB developers were asking for, including support for improved object modeling, inheritance, polymorphism, an expanded query language, and rich metadata for the specification of object/relational mapping.

But JPA's greatest achievement was not only to standardize ORM persistence technology for Java developers but also to incorporate the best practices for lightweight POJO based persistence (getting rid of the old and heavyweight Entity Bean architecture).

The JPA specification draws on ideas, concepts and standards from leading persistence frameworks such as TopLink, Hibernate, and Java Data Objects (JDO), as well as on earlier EJB container-managed persistence. But JPA itself is just a specification; a set of interfaces, and requires an implementation and a database to persist to.

Here's where Versant comes into play. The Versant team reviewed the state-of-the-art of JPA technology to bring to you Versant JPA, the JPA 2.0 provider for Versant Database technology.

In this multi-part article I will introduce the basic building blocks of the Java Persistence API as I walk you through the Versant JPA implementation from a user (developer) perspective.

Entities

An entity is basically a lightweight persistent domain object. The shape of an entity is defined in a persistence capable class (a.k.a. entity class) by typically adding a *@Entity* annotation (see *note 1*):

JPA Entity Class—Person

```
/* Copyright (C) 2011 Versant Inc.  http://www.versant.com */

package com.versant.jpa.tutorial.model;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    private long id;

    private String firstName;

    private String lastName;

    private Person() {

    }

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

}
```

Image 1: a JPA entity class is an ordinary Java class with the addition of an @Entity annotation

Traditionally the state of such object would be persisted to a table in a relational database while instances of such an entity would correspond to individual rows in the table. The necessary mapping information would have to be added to the mix (or defaulted by the JPA implementation) to match your domain object to tables.

Can you imagine getting rid of all the relational overhead while still being able to use JPA? Enter Versant JPA.

Note that in the *Person* example above we only need two annotations: *@Entity* and *@Id*. The *@Id* annotation designates the field to be used for the entity object identity in the datastore (or primary key in relational lingo). That's really all you need to make this an entity class using Versant JPA. Since Versant stores entity class instances as objects, no additional annotations specifying mapping information are needed.

The Persistence Unit

A JPA Persistence Unit is a logical grouping of user defined persistable classes with related settings such as the database connection:

persistence.xml

```
<persistence version="2.0">

   <persistence-unit name="jpa_tutorial_persistence_unit" transaction-type="RESOURCE_LOCAL">

        <class>com.versant.jpa.tutorial.model.Person</class>

        <class>com.versant.jpa.tutorial.model.Book</class>

        <properties>

            <property name="versant.connectionURL" value="jpa_tutorial@localhost" />

        </properties>

    </persistence-unit>

</persistence>
```

Image 2: a JPA persistence unit

The persistence unit lists the persistence capable classes (i.e. the entity classes) that make up your application's data model. These are the <*class*> elements you can see in the *persistence.xml* file above. The instances of these classes will be stored in the database defined by the connection setting.

Note: The persistence unit is defined in an XML file named *persistence.xml* which sits in the application's *META_INF* directory.

The Entity Manager

With JPA, the entity class objects in your application are managed by an *EntityManager*. An entity manager factory is responsible for entity managers. It is the factory that is associated with a persistence unit through a connection property to a Versant database.

The *EntityManager* interface basically provides the API for interacting with an Entity. An Entity can be in different states which are closely related to the *EntityManager*'s services:

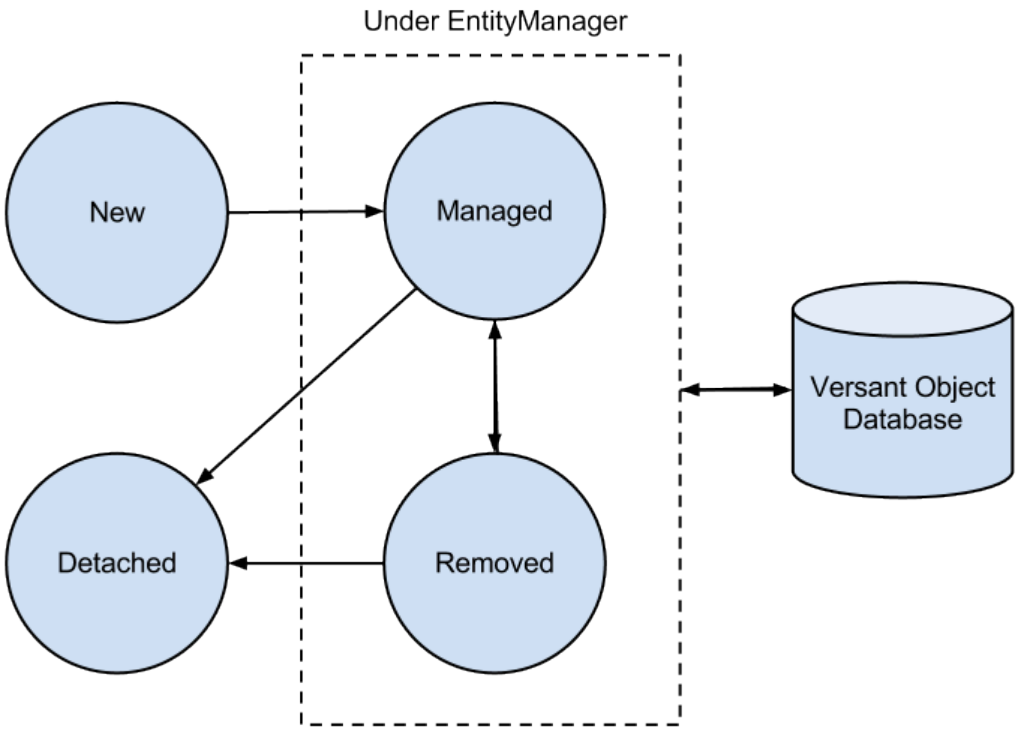

Image 3: Entity life-cycle in JPA

Entities created for the first time are in a state called "New" and they are not part of the database nor any EntityManager is aware of it. When the *EntityManager* performs a "persist" operation over an entity the entity's state changes to "Managed". This must be done in an active transaction (see Transactions below) which upon commit will effectively store the entity in the database.

Entities retrieved from the database (eg. via queries) are also in the "Managed" state so when they are modified within a transaction the changes will be saved to the database on the next commit.

The "remove" operation allows the *EntityManager* to mark an entity for deletion (state changes from "Managed" to "Removed" and the entity is deleted from the database upon commit).

Finally, entities can be completely disconnected from the *EntityManager* so they reach a state of "Detached" (eg. entities go into this state when its *EntityManager* is closed).

The most important services of the *EntityManager* API are:

- persist - Saves a new entity.
- merge - Updates the state of an entity into the database.
- detach - Disassociates an object from the database.
- remove - Removes the entity instance.

So, how do I start working with an *EntityManager* in Versant JPA?

The first thing is to create a factory and tell it which persistence unit to use for its configuration:

```
EntityManagerFactory emf =
       Persistence.createEntityManagerFactory("jpa_tutorial_persist~ ence_unit");
```

The resulting factory can now be called on to provide you with an EntityManager:

```
EntityManager em = emf.createEntityManager();
```

Transactions

All modifications on persistent objects, including creating new ones, are made in a transaction. A transaction is a unit of work performed by the client, your application, and the database server.

The *EntityManager* is responsible for the transaction. You can get access to the transaction with the manager's *getTransaction()* method. You work with a transaction by marking its beginning and end points with the methods *begin()* and *commit()*:

```
em.getTransaction().begin();
// do stuff with persistent objects . . .
em.getTransaction().commit();
```

Any changes to the entity objects in the transaction are written to the database (committed) when you call the transaction *commit()* method. If there's any contingency instead of *commit()*, you can call the transaction *rollback()* method to undo any changes you made.

When you create a new instance of an entity class, the *EntityManager* doesn't know if the instance should be persistent or not. So you have to tell the *EntityManager* that you want the instance to be persisted in the database by calling the entity manager *persist()* method for the new object instance (so your object ultimately gets into the "Managed" state).

Queries

Storing objects is of no value if you can't find them again. JPA uses the JPA Query Language (JPQL) which can be considered as an object oriented version of SQL. Users familiar with SQL should find JPQL very easy to learn and use.

A query is performed by asking the *EntityManager* for a *Query* instance. The desired query criteria, the predicate, is provided to the manager's *createQuery()* method.

```
Query query = em.createQuery("select p from Person p");
```

The query predicate (expressed as a string that follows the syntax of JPQL) simply selects all members from the entity class *Person*. The query is sent to the V/OD server and the result collection is returned:

```
List<Person> resultList = query.getResultList();
```

We can then use a loop to print the results:

```
for (Person person : resultList) {
      System.out.println(person);
}
```

or, alternatively, to remove all *Person* objects from the database:

```
for (Person person : resultList) {
      em.remove(person);
}
```

As you can see it's very easy and straight forward to get started with Versant JPA. If you have any previous experience with the Java Persistence API you'll feel right at home.

In part 2 of this series I will show you more advanced features such as cascading persistence (persistence-by-reachability), more advanced queries, change tracking and merging of detached objects.

If you want to try out a working example of all the features discussed here please go ahead and try Versant's JPA Technical Preview which is available as a free trial:

http://community.versant.com/jpa.aspx

Note 1: In general all JPA annotations can be replaced by defining tags in the persistence unit's ORM XML file.