Java Persistence API Demo

This document supports the source code of two different examples of utilization of the Java Persistence API. It assumes that you are already familiar with your favorite IDE. It includes three files:

- JPA_StudentExample.zip
- SqrRoot2EJB.zip
- SqrRoot2Client.zip

These are compressed Eclipse projects that you can import directly to Eclipse. However, they will not run before you correctly set all the libraries they require. Some of the configuration files are ready for JBoss. Nevertheless, you can also run these examples in NetBeans + GlassFish, with some additional effort.

Stand-alone Program

In our first example, we will use Java Persistence API (JPA) to access a simple database table. Using JPA annotations, we will set a correspondence between a Java Object of type `Student` and a database row in a table called `STUDENTS_TABLE`. Therefore, you should notice the annotations in the definition of the class `Student`. Although we use annotations for this, we could define a mapping between `Student` parameters and table columns, using an XML file instead. Annotations are simpler, although arguably less flexible than XML mapping: almost any change will force you to recompile the code.

Here, we show the code in the class `JPAWriteStudents`, which writes (persists) the objects to the database:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("TestPersistence");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
for (Student st : mylist)
  em.persist(st);
tx.commit();
```

Once we create an Entity Manger object, we can manage the boundaries of the transactions on our own (with `begin()` and `commit()`) and call the `persist()` method inside the transaction. One should notice that the name `TestPersistence` is an attribute of the XML element `persistent-unit` in the file `persistence.xml`.

To read the table we also include the class `JPAReadStudents`. It contains a very simple query in the Java Persistence Query Language. This language resembles SQL, the main difference being that it selects objects instead of table rows.

To compare the JPA solution with a simple SQL approach, we also include two additional classes `SQLReadStudents` and `SQLWriteStudents`. They are compatible with the previous two in the sense that the four classes can interact with each other using the same database table.

Apart from annotations, the remaining configuration lies in the `persistence.xml` file. This particular example works with Hibernate. You may need to change it slightly for other persistence middleware, like EclipseLink[1].

You may need to include a large number of libraries (we do not include any version, as this may change frequently):

- MySQL driver
- Hibernate libraries
- Slf4j
- Apache log4j

JPA from Enterprise Java Beans (EJBs)

In this case, we have an EJB and we will use JPA from there. This demo includes two different packages: one for the EJB and a second one that runs a small, stand-alone client for testing.

We will call `SqrRootBean` to our bean and we will name the interface `SqrRoot`. This EJB will simply compute a square root. Additionally, it will store the result in a database, to avoid repeating calculations it already made. Students should notice that this example serves only to introduce the configurations they need to do. Saving results of square roots in a database would certainly not save any time in practice.

In the EJB, the method `compute()` receives the number to square root. It first makes a query to the database, to check if it has computed that operation before. If it does not find an answer in the database, then it computes the square root and stores it. `ResultById()` is a helper method to give the `num`th entry in the database (i.e., according to the order in which the values entered the database). You can safely ignore it, if you want.

One should notice a few details in the EJB implementation. First, notice that we have no explicit transaction boundaries (`begin()` and `commit()`). We still have a running transaction that allows us to persist objects, but we do not manage it explicitly. Second, notice the existence of the annotation `@PersistenceContext` with the name "`TestPersistence`". As in the previous case, this name refers to the `persistence.xml` file. After construction of the EJB, the EJB container injects an

[1] Although the author of this text must say that he experienced intermittent problems with this same example in EclipseLink.

`EntityManager` instance to the private variable of our bean. One important point here in the file `persistence.xml` is the reference to the Java Transaction API source. You can either use the standard one, registered in JNDI under the name `java:/DefaultDS`, or define a custom one yourself. The default source uses the Hypersonic database, or use a specially configured one. We include an alternative `persistence.xml` configuration that refers to a different data source called `java:/MySqlDS`. This different jta data source allows you to use another database server that you may be more familiar with, in this case MySQL.

Remember that you need to set the same libraries in our project as before.

Good luck.