JDO Tutorial

by Travis Reeder, Thomas Mahler

Table of contents

1. Using the ObJectRelationalBridge JDO API

1.1. Introduction

This document demonstrates how to use ObjectRelationalBridge and the JDO API in a simple application scenario. The tutorial application implements a product catalog database with some basic use cases. The source code for the tutorial application is shipped in the `tutorials-src.jar` which can be downloaded here (http://www.apache.org/dyn/closer.cgi/db/ojb/) . The source for this tutorial is found in the directory **`org/apache/ojb/tutorial5`**.

This document is not meant as a complete introduction to JDO. For more information see: Sun's JDO site (http://java.sun.com/products/jdo/) .

> **Note:**
> OJB does not provide it's own JDO implementation yet. A full JDO implementation is in the scope of the 2.0 release.
> For the time being we provide a plugin to the JDO reference implementation called `OjbStore`. The `OjbStore` plugin resides in the package `org.apache.ojb.jdori.sql`.

1.2. Running the Tutorial Application

To install and run the demo application with the `ojb-blank` sample project (which is described in more detail here (../../docu/getting-started.html)) please follow the following steps:

1. Extract the `tutorial-src.jar` that you downloaded from here (http://www.apache.org/dyn/closer.cgi/db/ojb/) into the `src/java` subdirectory of the `ojb-blank` project.
   The JDO tutorial source files are contained in the `org/apache/ojb/tutorial5` subdirectory, and you can safely erase the subdirectories of the other tutorials.
2. Download the JDO Reference Implementation from Sun's JDO site (http://java.sun.com/products/jdo/) .
   Extract the archiv to a local directory and copy the files:
   - `jdori.jar`
   - `jdo.jar`

   into the `lib` directory of the project.
3. Now you can run the test application with these commands:

   ```
   ant build enhance-jdori
   ```

   from the toplevel project directory. The latter of these commands will enhance the jdo tutorial classes. Note that due to some limitations in the JDO reference implementation,

the ant target will only work for the JDO tutorial, so if you want to create you own JDO application using the ojb-blank project, you have to adapt the build file accordingly.
To setup the test database you can issue this command

```
ant setup-db
```

4. Now you can start the tutorial application by executing

```
cd build/resources

java org.apache.ojb.tutorial5.Main
```

from the project toplevel directory.

2. Using the JDO API in the UseCase Implementations

As shown here (../../index.html) OJB supports four different API's. The PersistenceBroker, the OTM layer, the ODMG implementation, and the JDO implementation.

The PB tutorial (../../docu/tutorials/pb-tutorial.html) implemented the sample application's use cases with the PersistenceBroker API. This tutorial will show how the same use cases can be implemented using the JDO API.

You can get more information about the JDO API at JDO javadocs (http://java.sun.com/products/jdo/javadocs/index.html) .

2.1. Obtaining the JDO PersistenceManager Object

In order to access the functionalities of the JDO API you have to deal with a special facade object that serves as the main entry point to all JDO operations. This facade is specified by the Interface `javax.jdo.PersistenceManager.`

A Vendor of a JDO compliant product must provide a specific implementation of the `javax.jdo.PersistenceManager` interface. JDO also specifies that a JDO implementation must provide a `javax.jdo.PersistenceManagerFactory` implementation that is responsible for generating `javax.jdo.PersistenceManager` instances.

So if you know how to use the JDO API you only have to learn how to obtain the OJB specific PersistenceManagerFactory object. Ideally this will be the only vendor specific operation.

In our tutorial application the `PersistenceManagerFactory` object is obtained in the constructor of the Application class and reached to the use case implementations for further usage:

```
public Application()
{
    factory = null;
    manager = null;
    try
    {
        // create OJB specific factory:
        factory = new OjbStorePMF();
    }
    catch (Throwable t)
    {
        System.out.println("ERROR: " + t.getMessage());
        t.printStackTrace();
    }
    useCases = new Vector();
    useCases.add(new UCListAllProducts(factory));
    useCases.add(new UCEnterNewProduct(factory));
    useCases.add(new UCEditProduct(factory));
    useCases.add(new UCDeleteProduct(factory));
    useCases.add(new UCQuitApplication(factory));
}
```

The class `org.apache.ojb.jdori.sql.OjbStorePMF` is the OJB specific `javax.jdo.PersistenceManagerFactory` implementation.

########## TODO: Put information about the .jdo files ############

The `PersistenceManagerFactory` object is reached to the constructors of the UseCases. These constructors store it in a protected attribute `factory` for further usage.

2.2. Retrieving collections

The next thing we need to know is how this Implementation instance integrates into our persistence operations.

In the use case `UCListAllProducts` we have to retrieve a collection containing all product entries from the persistent store. To retrieve a collection containing objects matching some criteria we can use the JDOQL query language as specified by the JDO spec. In our use case we want to select *all* persistent instances of the class Products. In this case the query is quite simple as it does not need any limiting search criteria.

We use the factory to create a PersistenceManager instance in step one. In the second step we ask the PersistenceManager to create a query returning all Product instances.

In the third step we perform the query and collect the results in a collection.

In the fourth step we iterate through the collection to print out each product matching our query.

```
public void apply()
{
    // 1. get a PersistenceManager instance
    PersistenceManager manager = factory.getPersistenceManager();
    System.out.println("The list of available products:");

    try
    {
        // clear cache to provoke query against database
        PersistenceBrokerFactory.
                    defaultPersistenceBroker().clearCache();

        // 2. start tx and form query
        manager.currentTransaction().begin();
        Query query = manager.newQuery(Product.class);

        // 3. perform query
        Collection allProducts = (Collection)query.execute();

        // 4. now iterate over the result to print each
        // product and finish tx
        java.util.Iterator iter = allProducts.iterator();
        if (! iter.hasNext())
        {
            System.out.println("No Product entries found!");
        }
        while (iter.hasNext())
        {
            System.out.println(iter.next());
        }
        manager.currentTransaction().commit();
    }
    catch (Throwable t)
    {
        t.printStackTrace();
    }
    finally
    {
        manager.close();
    }
}
```

2.3. Storing objects

Now we will have a look at the use case `UCEnterNewProduct`. It works as follows: first create a new object, then ask the user for the new product's data (productname, price and available stock). These data is stored in the new object's attributes. This part is no different from the PB tutorial (../../docu/tutorials/pb-tutorial.html) implementation. (Steps 1. and 2.)

Now we will store the newly created object in the persistent store by means of the JDO API. With JDO, all persistence operations must happen within a transaction. So the third step is to ask the PersistenceManager object for a fresh `javax.jdo.Transaction` object to work

with. The `begin()` method starts the transaction.

We then have to ask the PersistenceManager to make the object persistent in step 4.

In the last step we commit the transaction. All changes to objects touched by the transaction are now made persistent. As you will have noticed there is no need to explicitly store objects as with the PersistenceBroker API. The Transaction object is responsible for tracking which objects have been modified and to choose the appropriate persistence operation on commit.

```
public void apply()
{
    // 1. this will be our new object
    Product newProduct = new Product();
    // 2. now read in all relevant information and fill the new object:
    System.out.println("please enter a new product");
    String in = readLineWithMessage("enter name:");
    newProduct.setName(in);
    in = readLineWithMessage("enter price:");
    newProduct.setPrice(Double.parseDouble(in));
    in = readLineWithMessage("enter available stock:");
    newProduct.setStock(Integer.parseInt(in));

    // 3. create PersistenceManager and start transaction
    PersistenceManager manager = factory.getPersistenceManager();

    Transaction tx = null;
    tx = manager.currentTransaction();
    tx.begin();

    // 4. mark object as persistent
    manager.makePersistent(newProduct);

    // 5. commit transaction
    tx.commit();

    manager.close();
}
```

2.4. Updating Objects

The UseCase `UCEditProduct` allows the user to select one of the existing products and to edit it.

The user enters the products unique id. The object to be edited is looked up by this id. (Steps 1., 2. and 3.) This lookup is necessary as our application does not hold a list of all product objects.

The product is then edited (Step 4.).

In step five the transaction is commited. All changes to objects touched by the transaction are

now made persistent. Because we modified an existing object an update operation is performed against the backend database.

```
public void apply()
{
    PersistenceManager manager = null;

    // ask user which object should edited
    String in = readLineWithMessage("Edit Product with id:");
    int id = Integer.parseInt(in);

    Product toBeEdited;
    try
    {
        // 1. start transaction
        manager = factory.getPersistenceManager();
        manager.currentTransaction().begin();

        // We don't have a reference to the selected Product.
        // So we have to look it up first,


        // 2. Build a query to look up product by the id
        Query query = manager.newQuery(Product.class, "id == " + id);

        // 3. execute query
        Collection result = (Collection) query.execute();
        toBeEdited = (Product) result.iterator().next();

        if (toBeEdited == null)
        {
            System.out.println("did not find a matching instance...");
            manager.currentTransaction().rollback();
            return;
        }

        // 4. edit the existing entry
        System.out.println("please edit the product entry");
        in =
            readLineWithMessage(
                "enter name (was " + toBeEdited.getName() + "):");
        toBeEdited.setName(in);
        in =
            readLineWithMessage(
                "enter price (was " + toBeEdited.getPrice() + "):");
        toBeEdited.setPrice(Double.parseDouble(in));
        in =
            readLineWithMessage(
                "enter available stock (was "
                    + toBeEdited.getStock()
                    + "):");
        toBeEdited.setStock(Integer.parseInt(in));
```

```
        // 5. commit changes
        manager.currentTransaction().commit();
    }
    catch (Throwable t)
    {
        // rollback in case of errors
        manager.currentTransaction().rollback();
        t.printStackTrace();
    }
    finally
    {
        manager.close();
    }
}
```

2.5. Deleting Objects

The UseCase `UCDeleteProduct` allows the user to select one of the existing products and to delete it from the persistent storage.

The user enters the products unique id. The object to be deleted is looked up by this id. (Steps 1., 2. and 3.) This lookup is necessary as our application does not hold a list of all product objects.

In the fourth step we check if a Product matching to the id could be found. If no entry is found we print a message and quit the work.

If a Product entry was found we delete it in step 5 by calling the PersistenceManager to delete the persistent object. On transaction commit all changes to objects touched by the transaction are made persistent. Because we marked the Product entry for deletion, a delete operation is performed against the backend database.

```
public void apply()
{
    PersistenceManager manager = null;
    Transaction tx = null;
    String in = readLineWithMessage("Delete Product with id:");
    int id = Integer.parseInt(in);

    try
    {
        // 1. start transaction
        manager = factory.getPersistenceManager();
        tx = manager.currentTransaction();
        tx.begin();

        // 2. Build a query to look up product by the id
        Query query = manager.newQuery(Product.class, "id == " + id);

        // 3. execute query
```

```
        Collection result = (Collection) query.execute();

        // 4. if no matching product was found, print a message
        if (result.size() == 0)
        {
            System.out.println("did not find a Product with id=" + id);
            tx.rollback();
            manager.close();
            return;
        }
        // 5. if a matching product was found, delete it
        else
        {
            Product toBeDeleted = (Product) result.iterator().next();
            manager.deletePersistent(toBeDeleted);
            tx.commit();
            manager.close();
        }
    }
    catch (Throwable t)
    {
        // rollback in case of errors
        //broker.abortTransaction();
        tx.rollback();
        t.printStackTrace();
    }
}
```

3. Conclusion

In this tutorial you learned to use the standard JDO API as implemented by the OJB system within a simple application scenario. I hope you found this tutorial helpful. Any comments are welcome.