Persistence Broker Tutorial

by Brian McCallister

1. The PersistenceBroker API

1.1. Introduction

The PersistenceBroker API provides the lowest level access to OJB's persistence engine. While it is a low-level API compared to the OTM, ODMG, or JDO API's it is still very straightforward to use.

The core class in the PersistenceBroker API is the `org.apache.ojb.broker.PersistenceBroker` class. This class provides the point of access for all persistence operations in this API.

This tutorial operates on a simple example class:

```
package org.apache.ojb.tutorials;

public class Product
{
    /* Instance Properties */

    private Double price;
    private Integer stock;
    private String name;lean

    /* artificial property used as primary key */

    private Integer id;

    /* Getters and Setters */
    ...
}
```

The metadata descriptor for mapping this class is described in the mapping tutorial

The source code for this tutorial is available with the source distribution of OJB in the `src/test/org/apache/ojb/tutorials/` directory.

1.2. A First Look - Persisting New Objects

The most basic operation is to persist an object. This is handled very easily by just

1. obtaining a `PersistenceBroker`
2. begin the PB-transaction
3. storing the object via the `PersistenceBroker`
4. commit transaction
5. closing the `PersistenceBroker`

For example, the following function stores a single object of type `Product`.

```
public static void storeProduct(Product product)
{
    PersistenceBroker broker = null;
    try
```

```
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        broker.store(product);
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
}
```

Two OJB classes are used here, the `PersistenceBrokerFactory` and the `PersistenceBroker`. The `PersistenceBrokerFactory` class manages the lifecycles of `PersistenceBroker` instances: it creates them, pools them, and destroys them as needed. The exact behavior is very configurable.

In this case we used the static `PersistenceBrokerFactory.defaultPersistenceBroker()` method to obtain an instance of a `PersistenceBroker` to the default data source. This is most often how it is used if there is only one database for an application. If there are multiple data sources, a broker may be obtained by name (using a `PBKey` instance as argument in `PersistenceBrokerFactory.createPersistenceBroker(pbKey)`).

It is worth noting that the `broker.close()` call is made within a `finally {...}` block. This ensures that the broker will be closed, and returned to the broker pool, even if the function throws an exception.

To use this function, we just create a `Product` and pass it to the function:

```
Product product = new Product();
product.setName("Sprocket");
product.setPrice(1.99);
product.setStock(10);
storeProduct(product);
```

Once a `PersistenceBroker` has been obtained, its `PersistenceBroker.store(Object)` method is used to make an object persistent.

Maybe you have noticed that there has not been an assignment to `product.id`, the primary-key attribute. Upon storing `product` OJB detects that the attribute is not properly set and assigns a unique id. This automatic assignment of unique Ids for the attribute `id` has been explicitly declared in the XML repository file, as we discussed in the .

If several objects need to be stored, this can be done within a transaction, as follows.

```
public static void storeProducts(Product[] products)
{
    PersistenceBroker broker = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        for (int i = 0; i < products.length; i++)
        {
            broker.store(products[i]);
        }
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
```

```
    {
        if (broker != null) broker.close();
    }
}
```

This contrived example stores all of the passed Product instances within a single transaction via the `PersistenceBroker.beginTransaction()` and `PersistenceBroker.commitTransaction()`. These are database level transactions, not object level transactions.

1.3. Querying Persistent Objects

Once objects have been stored to the database, it is important to be able to get them back. The PersistenceBroker API provides two mechanisms for building queries, by using a template object, or by using specific criteria.

```
public static Product findByTemplate(Product template)
{
    PersistenceBroker broker = null;
    Product result = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        QueryByCriteria query = new QueryByCriteria(template);
        result = (Product) broker.getObjectByQuery(query);
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return result;
}
```

This function finds a `Product` by building a query against a template `Product`. The template should have any properties set which should be matched by the query. Building on the previous example where a product was stored, we can now query for that same product:

```
Product product = new Product();
product.setName("Sprocket");
product.setPrice(new Double(1.99));
product.setStock(new Integer(10));
storeProduct(product);

Product template = new Product();
template.setName("Sprocket");
Product sameProduct = findByTemplate(template);
```

In the above code snippet, `product` and `sameProduct` will reference the same object (assuming there are no additional products in the database with the name "Sprocket").

The template `Product` has only one of its properties set, the `name` property. The others are all null. Properties with null values are not used to match.

An alternate, and more flexible, way to have specified a query via the PersistenceBroker API is by constructing the criteria on the query by hand. The following function does this.

```
public static Collection getExpensiveLowStockProducts()
{
    PersistenceBroker broker = null;
    Collection results = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();

        Criteria criteria = new Criteria();
        criteria.addLessOrEqualThan("stock", new Integer(20));
```

```
        criteria.addGreaterOrEqualThan("price", new Double(100000.0));

        QueryByCriteria query = new QueryByCriteria(Product.class, criteria);
        results = broker.getCollectionByQuery(query);
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return results;
}
```

This function builds a `Criteria` object and uses it to set more complex query parameters - in this case greater-than and less-than contraints. Looking at the first constraint put on the criteria, `criteria.addLessOrEqualThan("stock", new Integer(10));` notice the arguments. The first is the property name on the object being queried for. The second is an `Integer` instance to be used for the comparison.

After the `Criteria` has been built, the `QueryByCriteria` constructor used is also different from the previous example. In this case the criteria does not know the type of the object it is being used against, so the `Class` must be specified to the query.

Finally, notice that this example uses the `PersistenceBroker.getCollectionByQuery(...)` method instead of the `PersistenceBroker.getObjectByQuery(...)` method used previously. This is used because we want all of the results. Either form can be used with either method of constructing queries. In the case of the `PersistenceBroker.getObjectByQuery(...)` style query, the first matching object is returned, even if there are multiple matching objects.

1.4. Updating Persistent Objects

The same mechanism, and method, is used for updating persistent objects as for inserting persistent objects. The same `PersistenceBroker.store(Object)` method is used to store a modified object as to insert a new one - the difference between new and modified objects is irrelevent to OJB.

This can cause some confusion for people who are very used to working in the stricter confines of SQL inserts and updates. Basically, OJB will insert a new object into the relational store if the primary key, as specified in the O/R metadata is not in use. If it is in use, it will update the existing object rather than create a new one.

This allows programmers to treat every object the same way in an object model, whether it has been newly created and made persistent, or materialized from the database.

Typically, making changes to a peristent object first requires retrieving a reference to the object, so the typical update cycle, unless the application caches objects, is to query for the object to modify, modify the object, and then store the object. The following function demonstrates this behavior by "selling" a Product.

```
public static boolean sellOneProduct(Product template)
{
    PersistenceBroker broker = null;
    boolean isSold = false;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        QueryByCriteria query = new QueryByCriteria(template);
        Product result = (Product) broker.getObjectByQuery(query);

        if (result != null)
        {
            broker.beginTransaction();
            result.setStock(new Integer(result.getStock().intValue() - 1));
            broker.store(result);
            // alternative, more performant
            // broker.store(result, ObjectModificationDefaultImpl.UPDATE);
```

```
            broker.commitTransaction();
            isSold = true;
        }
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return isSold;
}
```

This function uses the same query-by-template and `PersistenceBroker.store()` API's examined previously, but it uses the store method to store changes to the object it retrieved. It is worth noting that the entire operation took place within a transaction.

1.5. Deleting Persistent Objects

Deleting persistent objects from the repository is accomplished via the `PersistenceBroker.delete()` method. This removes the persistent object from the repository, but does not affect any change on the object itself. For example:

```
public static void deleteProduct(Product product)
{
    PersistenceBroker broker = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();
        broker.beginTransaction();
        broker.delete(product);
        broker.commitTransaction();
    }
    catch(PersistenceBrokerException e)
    {
        if(broker != null) broker.abortTransaction();
        // do more exception handling
    }
    finally
    {
        if (broker != null) broker.close();
    }
}
```

This method simply deletes an object from the database.

2. Notes on Using the PersistenceBroker API

2.1. Pooling PersistenceBrokers

The `PersistenceBrokerFactory` pools `PersistenceBroker` instances. Using the `PersistenceBroker.close()` method releases the broker back to the pool under the default implementation. For this reason the examples in this tutorial all retrieve, use, and close a new broker for each logical transaction.

2.2. Transactions

Transactions in the PeristenceBroker API are database level transactions. This differs from object level transactions. The broker does not maintain a collection of modified, created, or deleted objects until a commit is called -- it operates on the database using the databases transaction mechanism. If object level transactions are required, one ofthe higher level API's

(ODMG, JDO, or OTM) should be used.

2.3. Exception Handling

Most `PersistenceBroker` operations throw a `org.apache.ojb.broker.PersistenceBrokerException`, which is derived from `java.lang.RuntimeException` if an error occurs. This means that no try/catch block is **required** but does not mean that it should not be used. This tutorial specifically does not catch exceptions all in order to focus more tightly on the specifics of the API, however, best usage would be to include a try/catch/finally block around persistence operations using the PeristenceBroker API.

Additionally, the closing of `PersistenceBroker` instances is best handled in `finally` blocks in order to guarantee that it is run, even if an exception occurs. If the `PersistenceBroker.close()` is not called then the application will leak broker instances. The best way to ensure that it is always called is to always retrieve and use `PersistenceBroker` instances within a `try {...}` block, and always close the broker in a `finally {...}` block attached to the `try {...}` block.

A better designed `getExpensiveLowStockProducts()` method is presented here.

```
public static Collection betterGetExpensiveLowStockProducts()
{
    PersistenceBroker broker = null;
    Collection results = null;
    try
    {
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();

        Criteria criteria = new Criteria();
        criteria.addLessOrEqualThan("stock", new Integer(20));
        criteria.addGreaterOrEqualThan("price", new Double(100000.0));

        QueryByCriteria query = new QueryByCriteria(Product.class, criteria);
        results = broker.getCollectionByQuery(query);
    }
    catch (PersistenceBrokerException e)
    {
        // Handle exception
    }
    finally
    {
        if (broker != null) broker.close();
    }
    return results;
}
```

Notice first that the `PersistenceBroker` is retrieved and used within the confines of a `try {...}` block. Assuming nothing goes wrong the entire operation will execute there, all the way to the `return results;` line. Java guarantees that `finally {...}` blocks will be called before a method returns, so the `broker.close()` method is only included once, in the `finally` block. As an exception may have occured while attempting to retrieve the broker, a not-null test is first performed before closing the broker.