Setting Up Hibernate Entity Manager

For Easy Storage and Retrieval of Java Objects using Object Relational Mapping

James Rundquist

Upon deciding to use Hibernate to enable persistence in your java project, there are many steps that then must be taken before your application can even begin communicating with your database. The hardest and most confusing of these has to be the configuration of a Hibernate object manager. In this document we will cover the setup necessary to get Hibernate working using the entity manager configuration, in which object are mapped to the database and each other through annotations in the java code.

Entity Manager Path

Hibernate can operate in either one of two 'modes', either through an EntityManager, or a SessionManager. The differences between the modes are small as far as the usage of the managers is concerned (actually retrieving and saving objects), the major differences are in how the objects are told to map to the database, and in the database configuration itself. The entity manager allows for mapping to be specified in the object's code through the use of annotations, while the SessionManager requires separate Object.hbm.xml file to specify the mapping of a given object.

This document will cover the use of the EntityManager. This setup allows you to identify objects as database entities, map them to specific tables, set the mapping characteristics of a given field, and even specify relationships between objects all from the same .java file the object itself is defined in. In this way, the code related to a given object is all contained within the object code itself.

Hibernate Libraries

Make a special note to insure all the necessary Hibernate libraries are included in your class path. If some obscure library needed by Hibernate is missing, or for any reason not the library Hibernate was expecting, it can have a tendency to implode, leaving you wondering if it was your code that killed it, the configuration you specified, or hibernate itself. So, remember to check the Hibernate download page to insure you have the most recent packages.

Persistence.xml

After spending countless hours fumbling through the vague and incomplete Hibernate online documentation, we found that when using the EntityManager you **DO NOT NEED** the hibernate.cfg.xml file the documentation suggests you do, and if for some reason you do, our program seemed to work fine without it… so it can't be too important.

A file that is necessary is the *persistence.xml* file. This XML file configures Hibernate to work with the database you have previously set up. The file must be located in the */META-INF/* directory of the project. Note that this directory must be copied into your build directory, or linked to it, in order for Hibernate to be able to find it while building. The *persistence.xml* file specifies information such as, how Hibernate should build the table, to what database it should work in, connection credentials, what Dialect to use, and the Driver necessary to communicate with the server. For convenience, the *persistence.xml* file used in our project has been included (bolded items must be configured before it will run).

One property that may be of importance is that of *hibernate.hbm2ddl.auto*. This property specifies how hibernate handles the database tables that the objects you will be saving are to be stored in. My

group in particular had many issues with this before we discovered what exactly it was doing. When this property is set improperly, it can completely drop your tables and recreate them every time a new connection is made to the database. Possible values for this property and their meanings are listed below.

- *validate*: validate the schema, makes no changes to the database.
- *update*: update the schema.
- *create*: creates the schema, destroying previous data.
- *create-drop*: drop the schema at the end of the session.

In our project we used the update method because we were often making changes to the objects and their relations, and did not want to update the schema ourselves. For a testing setup create-drop would probably be the most advisable, while it would take additional time in creating the schema every run, it would insure that old test data would not interfere with the new, and currently running, tests.

A very important note is that the persistence.xml file may define more than one configuration. The included example shows how this is done, by simply defining more than one <persistence-unit> tag in the file. Defining more than one configuration is very useful when it comes to writing JUnit tests, allowing the same code to interface with a different database, separating test data from production data.

Creating an EntityManager

Once you have defined the configuration for your project, and insured the proper libraries are included in your class path (and that persistence.xml is in the /build/META-INF/ folder, assuming /build/ is your build directory), you can begin creating entity managers. I recommend creating a singleton class that in turn has only one entity manager because the speed of your application will be greatly reduced if a new connection is made to the database any time the user requests new information.

The code below shows code from my team's DataBaseAccessor (DBA) that connected to the database using a new entity manager. Note the fact that the DBA has an optional parameter that is passed to the createEntityManagerFactory() method that defaults to "domain". This String parameter specifies the named configuration in the *persistence.xml* file created earlier. Thus if you wanted to create a new DBA for the testing database, simply call DatabaseAccessor("testing") which would, upon connecting to the database, connect to the testing database.

Code 1: A snippet of code showing the creation of an EntityManager

```
private EntityManagerFactory emf = null;
private EntityManager manager = null;
private String persistence_unit = "domain";

public DataBaseAccessor(){
}

public DataBaseAccessor(String unit){
    this.persistence_unit = unit;
}

public void connectToDatabase(){
    if ( !isConnected() ){
        try {
            this.emf = Persistence.createEntityManagerFactory(this.persistence_unit);
            // Retrieve an application managed entity manager
            this.manager = this.emf.createEntityManager();
        }catch(Throwable t){
            // Handle the error thrown
        }
    }
}
```

Also note that for clarity's sake the error handling has been simplified in the above code, in a real implementation, it would be advisable to handle errors that could occur when the database connection fails.

A Quick Note on Finding and Saving

Once you have configured Hibernate and presumably connected to the database successfully, you are now ready to read through how to mark objects as a persist-able entity using the @Entity notation. After marking the classes you need to persist, you are ready to actually save and load them from your database. I personally recommend creating save(), update(), and find() methods within your DatabaseAccessor class. This will abstract away the entity manager from the end coder, allowing for a change in the entity manager handling without affecting the code. Example methods are shown below.

Code 2 Example save, update, and find methods

```
public boolean save(Object objectToSave) {
    this.em.getTransaction().begin();
    this.em.persist(objectToSave);
    this.em.getTransaction().commit();
    return true;
}

public boolean update(Object objectToSave) {
    this.em.getTransaction().begin();
    this.em.merge(objectToSave);
    this.em.getTransaction().commit();
    return true;
}

public <T> T find(Class<T> arg0, Object arg1){
    this.em.getTransaction().begin();
    T found = this.em.find(arg0, arg1);
    this.em.getTransaction().commit();
    return found;
}
```

Note how the methods all include the getTransaction.begin() and getTransaction.commit() methods. These insure that the object being found or saved is saved to the database at this moment, through a transaction. A transaction acts as one unit of work. If something within the transaction fails, for example linking two objects together, the entire transaction will be rolled back (if possible) and nothing will be committed to the database.

Example Persistence.xml File ::

```
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit name="domain">
        <properties>
            <property name="hibernate.archive.autodetection" value="class, hbm"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.connection.driver_class"
                                value="[com.mysql.jdbc.Driver]"/>
            <property name="hibernate.connection.url"
                                value="jdbc:mysql://example.com/my_database"/>
            <property name="hibernate.connection.username" value="[USERNAME]"/>
            <property name="hibernate.connection.password" value="[PASSWORD]"/>
            <property name="hibernate.dialect"
                                value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
             <property name="hibernate.c3p0.min_size" value="5"/>
            <property name="hibernate.c3p0.max_size" value="20"/>
            <property name="hibernate.c3p0.timeout" value="300"/>
            <property name="hibernate.c3p0.max_statements" value="50"/>
            <property name="hibernate.c3p0.idle_test_period" value="3000"/>
        </properties>
    </persistence-unit>
    <persistence-unit name="testing">
        <properties>
            <property name="hibernate.archive.autodetection" value="class, hbm"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.connection.driver_class"
                                value="[com.mysql.jdbc.Driver]"/>
            <property name="hibernate.connection.url"
                                value="jdbc:mysql://example.com/my_testing_database"/>
            <property name="hibernate.connection.username" value="[USERNAME]"/>
            <property name="hibernate.connection.password" value="[PASSWORD]"/>
            <property name="hibernate.dialect"
                                value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
                      <property name="hibernate.c3p0.min_size" value="5"/>
            <property name="hibernate.c3p0.max_size" value="20"/>
            <property name="hibernate.c3p0.timeout" value="300"/>
            <property name="hibernate.c3p0.max_statements" value="50"/>
            <property name="hibernate.c3p0.idle_test_period" value="3000"/>
        </properties>
    </persistence-unit>
</persistence>
```

Useful Links and Notes

Hibernate Documentation Page
http://www.hibernate.org/docs

Hibernate Annotation Guide
http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/

Google – Your source for everything else
http://google.com/

A Helpful O'Reilly Book : Harnessing Hibernate
http://oreilly.com/catalog/9780596517724?green=24000004332&cmp=af-mybuy-9780596517724.IP

Skill-Guru Getting Started Guide - Very Helpful
http://www.skill-guru.com/blog/2010/06/15/hibernate-search-%E2%80%93-getting-started/

Notes

When using @Entity import javax.persistence.Entity NOT org.hibernate.annotation.Entity.
On that note, always prefer the javax.presisence package to the org.hibernate.annotation package, if given the choice.

When defining your objects as Entities, test them one by one.
If one has an issue, it will cause other objects that relate to it to have issues, and you may have some trouble reading the Hibernate exceptions to find out exactly what is going on.

In general expect to spend a lot of time setting up and debugging Hibernate.
But, after it is all set up, it will make persistence in java SO much easier than having to manually maintain and map your objects to your databases. In all, it is a great experience and a wonderful tool and all teams should seriously consider looking into using it.