Get started quickly with Hibernate Annotations and JPA 2

Get started quickly with Hibernate Annotations and JPA 2

Aug 21, 2012 1:08 PM, **Lincoln Baxter III** [Original post]

Chapter 1 – Step By Step

Getting started with Hibernate and JPA (Java Persistence API) can be tricky, but this step-by-step tutorial explains exactly what needs to be done to set up your application to use this technology. This chapter covers very basic mapping and persistence.

When we are finished with this tutorial, we will have a standalone Java SE application with database connectivity. This article is part of a series: Guide to Hibernate Annotations .

Add dependencies to your Maven POM

Or follow the official Hibernate getting started guide in order to configure a non-maven project.

pom.xml Or download the entire demo

```
<dependencyManagement>
      <dependencies>
         <dependency>
            <groupId>org.jboss.spec</groupId>
            <artifactId>jboss-javaee-6.0</artifactId>
            <version>3.0.1.Final</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
         <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.1.6.Final</version>
         </dependency>
         <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.2.8</version>
         </dependency>
         <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
            <scope>test</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>

   <dependencies>
      <dependency>
         <groupId>org.hibernate.javax.persistence</groupId>
         <artifactId>hibernate-jpa-2.0-api</artifactId>
         <scope>provided</scope>
      </dependency>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-entitymanager</artifactId>
      </dependency>
      <dependency>
         <groupId>org.hsqldb</groupId>
         <artifactId>hsqldb</artifactId>
      </dependency>
```

```
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
      </dependency>
   </dependencies>
```


Create a database connection

Before we can start hammering away at the database, there are a few things we need to take care of. First, we'll need to set up a `persistence.xml` file, then retrieve an instance of an EntityManager, which is a JPA construct we'll describe more later.

Set up persistence.xml

This is my least-favorite task of using JPA and Hibernate, and the reason is because I can never remember how to actually configure persistence.xml, so for this demo I turned to JBoss Forge 's persistence plugin to help me get things set up. You, however, can just copy this file.

You can always do this configuration in Java if you are using pure Hibernate, but the standard approach is to use persitence.xml.
META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="hsqldb-ds" transaction-type="RESOURCE_LOCAL">

      <description>HSQLDB Persistence Unit</description>
      <provider>org.hibernate.ejb.HibernatePersistence</provider>

<!-- This is where we tell JPA/Hibernate about our @Entity objects -->      <class>org.ocpsoft.example.hibernate.model.SimpleObject</class>

      <properties>
         <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:demodb" />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <property name="javax.persistence.jdbc.password" value="" />
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
         <property name="hibernate.hbm2ddl.auto" value="create-drop" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />
         <property name="hibernate.transaction.flush_before_completion" value="true" />
      </properties>
   </persistence-unit>
</persistence>
```

Get an EntityManager

`EntityManagerUtil` is a convenience class that provides your application access to the active `EntityManagerFactory` . The `EntityManagerFactory` controls connections to the database, and also gives you access to `EntityManager` instances.

`EntityManager` is the API you'll be using to actually interact with the database via SQL or JQL (JPA Query Language.)

You should note that we are also using the `javax.persistence.Persistence` class to obtain an `EntityManagerFactory`

EntityManagerUtil.java

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EntityManagerUtil
{
   public static EntityManagerFactory getEntityManagerFactory()
   {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("hsqldb-ds");
      return emf;
   }
}
```

Configure logging

Fortunately for us, Hibernate comes with a default logging configuration that shows us pretty much what we need, and if

you are using it on a Java EE Application Server like JBoss AS or GlassFish then this logging is configured in the server itself; however, if you want to customize the log output for our standalone Java SE demo, you'll need to take a look at jboss-logging .

Create an @Entity – SimpleObject.java

It's time to create our first object. Notice that this is a simple POJO with two fields: id and version. The *id* field is a required field, but before you complain, there is a philosophy of database design that dictates every entity table should have its own *id* column; some disagree with this, but I've found it a very convenient practice. Try it out before you pass judgment.

In order to map an object, a few things always need to be done:

- The class must be annotated with `@javax.persistence.Entity`
- The class must have a `@javax.persistence.Id` column
- The class must have a default (or undefined) constructor method. This means it must have a constructor that has no arguments.
- The `@javax.persistence.Version` column is how Hibernate performs optimistic locking . If a record has been modified by another transaction, your transaction's version field will be out of date, and a `javax.persistence.OptimisticLockException` exception will be thrown.
- By default, all fields in your class will be persisted to the database if possible. If there is a field that should not be recorded in the database, mark that field with the `@javax.persistence.Transient` annotation.
- To add new columns in the database, simply add new fields in your object.

And now we need to create the object itself. Notice that we have not implemented the `.equals()` and `.hashCode()` methods; these methods will need to be implemented in order to ensure proper object equality and consistency between the Hibernate/JPA and our application. Without a proper implementation of these two methods, bad things can happen like duplicate rows being added, strange or missing data from result sets.

For more information about implementing `.equals()` and `.hashCode()` please read this Stack Overflow entry .
SimpleObject.java

```
@Entity
@Table
public class SimpleObject implements Serializable
{
   private static final long serialVersionUID = -2862671438138322400L;

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = "id", updatable = false, nullable = false)
   private Long id = null;

   @Version
   @Column(name = "version")
   private int version = 0;

   public Long getId()
   {
      return this.id;
   }

   public void setId(final Long id)
   {
      this.id = id;
   }

   public int getVersion()
   {
      return this.version;
   }

   public void setVersion(final int version)
   {
      this.version = version;
   }
}
```

Put it all together

Our driver class is very simple; it is a JUnit test case that will:

- Get a handle to JPA EntityManager (This is equivalent to the Hibernate Session)
- Create and persist two new *SimpleObjects*

- Print out the generated IDs and assert that they are as expected.

HibernateDemoTest.java

```
package org.ocpsoft.example.hibernate;

import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;

import junit.framework.Assert;

import org.junit.*;
import org.ocpsoft.example.hibernate.model.SimpleObject;
import org.ocpsoft.example.hibernate.util.EntityManagerUtil;

/**
 * @author <a href="mailto:lincolnbaxter@gmail.com">Lincoln Baxter, III</a>
 */
public class HibernateDemoTest
{
   private EntityManager em;

   @Before
   public void beforeEach()
   {
      em = EntityManagerUtil.getEntityManagerFactory().createEntityManager();
   }

   @After
   public void afterEach()
   {
      em.close();
   }

   @Test
   public void testAutoIncrement()
   {
      EntityTransaction transaction = em.getTransaction();
      transaction.begin();

      SimpleObject object0 = new SimpleObject();
      SimpleObject object1 = new SimpleObject();

      // IDs start as null
      Assert.assertEquals((Long) null, object0.getId());
      Assert.assertEquals((Long) null, object1.getId());

      em.persist(object0);
      em.persist(object1);

      transaction.commit();

      System.out.println("Object 0");
      System.out.println("Generated ID is: " + object0.getId());
      System.out.println("Generated Version is: " + object0.getVersion());

      System.out.println("Object 1");
      System.out.println("Generated ID is: " + object1.getId());
      System.out.println("Generated Version is: " + object1.getVersion());

      Assert.assertEquals((Long) 1l, object0.getId());
      Assert.assertEquals((Long) 2l, object1.getId());
   }
}
```

And here we are. When you run HibernateDemoTest, you should have a fully functional Hibernate enabled HSQL database with one table. You should see Hibernate create the table, then execute our queries. The application will then print out our results – play around with adding new fields to your object, take a look at the console output. I think you will quickly see how things work.

Your output should look something like this:

```
Object 0
Generated ID is: 1
Generated Version is: 0
Object 1
Generated ID is: 2
Generated Version is: 0
```

A few more things to remember

Take your time. Don't give up. Ask questions. *READ the forums and documentation.*

Hibernate and JPA are BIG. They are very powerful, very complex tools. It took me a long time to get even a simple object relationship working the way I wanted to. The learning curve is not small, and you will probably be frustrated for a while. I have, however, come to the point where I see so much benefit, that I feel it was completely worth my time to learn.

Configuration is **everything** . Hibernate can do anything you want: You can use it to tie in to existing database schema; you can use it to run plain SQL queries or stored procedures, but you need to make sure that you have a solid understanding of the configuration. If you start running into problems, chances are that you are missing something small.

Additional reading

If you want more information about Hibernate, Hibernate Annotations, or Hibernate Search, I recommend reading a section in another OCPsoft blog about using extended PersistenceContexts . You can also try some books written by my colleagues and peers. (Note, you will need to disable any ad-block programs in order to see these links.)

Want more information? This article is part of a series: Guide to Hibernate Annotations .