Introduction to Java Persistence with Hibernate

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

JDBC is Java's solution for encapsulating the DBMS from the application code. However, using it requires developers to write SQL queries and exposes the data model to the application code. JDO is an effort to fully abstract and encapsulate the database from the application code and it realizes that through byte code enhancement. One alternate solution that is gaining popularity is Hibernate. In this article we present an introduction to the Hibernate open source persistence framework using a simple example.

A Persistence Example

Let's start with a simple example. We will deal with two classes: Person and Dog. The Dog class (Dog.java) is first shown below:

```
package com.agiledeveloper;

public class Dog
{
      private long id;
      private String name;
      private Person friend;

      public Dog() {}

      public long getId() { return id; }
      public void setId(long theID) { id = theID; }

      public String getName() { return name; }
      public void setName(String newName) { name = newName; }

      public Person getFriend() { return friend; }
      public void setFriend(Person newFriend) { friend = newFriend; }
}
```

A Dog has an id, a name and reference to a Person who is the Dog's friend. It also has getters and setters for each of the fields.

The Person class (Person.java) is next shown below:

```
package com.agiledeveloper;

import java.util.*;

public class Person
{
      private long id;
      private String firstName;
      private String lastName;
      private Set pets = new HashSet();
```

```
	public Person() {}

	public long getId() { return id; }
	public void setId(long theNewId) { id = theNewId; }

	public String getFirstName() { return firstName; }
	public void setFirstName(String newFirstName)
				{ firstName = newFirstName; }

	public String getLastName() { return lastName; }
	public void setLastName(String newLastName)
				{ lastName = newLastName; }

	public Set getPets() { return pets; }
	public void setPets(Set thePets) { pets = thePets; }

	public void addPet(Dog aPet)
	{
		if (!pets.contains(aPet))
		{
			pets.add(aPet);
		}
	}

	public void removePet(Dog aDog)
	{
		pets.remove(aDog);
	}
}
```

The Person class has an id, the first and last name followed by the set of pets. Methods of the Person class (addPet, getPet, setPets, removePet) will allow us to manage the Dogs that a Person may have as Pets.

Looking at the code above, there is no relevant code as far as persistence is concerned. The code to persist is kept separate in a mapping file. Let's first take a look at the Dog.hbm.xml file:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
	<class name="com.agiledeveloper.Dog" table="dog">
		<id name="id" column="id" type="long" unsaved-value="0">
			<generator class="native" />
		</id>
		<property name="name" column="name" type="string"/>
	    <many-to-one name="friend"
		  class="com.agiledeveloper.Person" column="person_id" />
	</class>
</hibernate-mapping>
```

This file maps the Java class to the persistence layer. It says that an object of Dog class will be stored in a dog table. Further, it says that the id field and the name field of Dog maps to the id column and the name column in the dog table, respectively. It then defines the relationship between the Person and the Dog, though the friend field as a many-to-one relationship to the persistent column named person_id which acts as a foreign key.

Let's now take a look at the Person.hbm.xml mapping file:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
      <class name="com.agiledeveloper.Person" table="person">
            <id name="id" column="id" type="long" unsaved-value="0">
                  <generator class="native" />
            </id>
            <property name="firstName"
                        column="first_name" type="string"/>
            <property name="lastName"
                        column="last_name" type="string"/>
            <set name="pets" cascade="all" inverse="true" lazy="true">
                  <key column="person_id"/>
                  <one-to-many class="com.agiledeveloper.Dog"/>
            </set>
      </class>
</hibernate-mapping>
```

This file maps the Person class to the Person table. Apart from the straight forward mapping of the id, firstName and lastName fields, it also defines the set of pets as a one-to-many relationship to the objects of Dog class. Of special interest is the value of the lazy attribute. A value of true indicates that when a Person object is fetched from the database into memory, the related pets (Dog objects) need not be fetched. These are fetched if and only if requested. This on demand fetch may be desirable under situations where not all related objects are used or accessed in an application scenario.

The above two mapping files help us map the objects to the database tables. However, which database do we connect to? This information is provided in yet another configuration or property file named here as hibernate.properties:

```
hibernate.connection.username = CONNECTIONUSER
hibernate.connection.password = PasswordForCONNECTIONUSER
hibernate.connection.driver_class =
                        DRIVER_FOR_DATABASE
hibernate.connection.url
       URL_FOR_CONNECTION_TO_DB
hibernate.dialect = HIBERNATE_CLASS_FOR_DIALECT
hibernate.show_sql = true
```

The hibernate.show_sql is useful for debugging purpose. As the program executes, you can take a look at the generated SQL. You may use any relational database of your choice like Oracle, MySql, SQL Server, etc. In this example, I will use the SQL Server. The hibernate.properties file modified to use SQL Server is shown below:

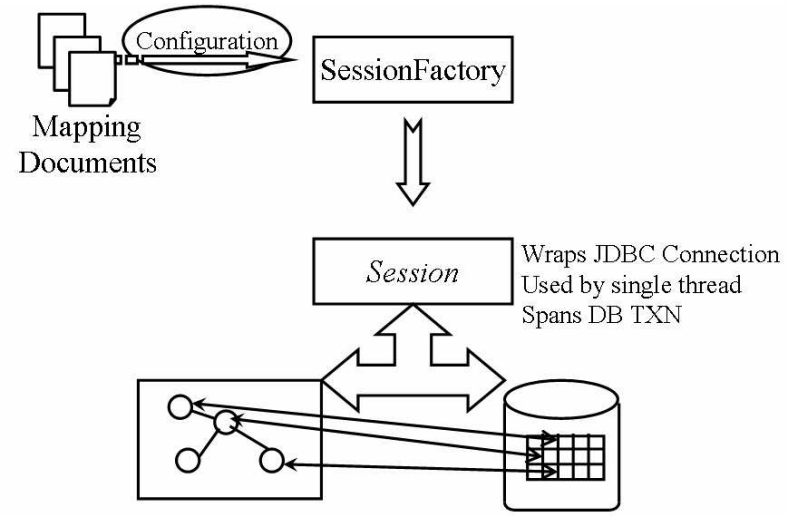
```
hibernate.connection.username = sa
hibernate.connection.password = PasswordForSA
hibernate.connection.driver_class =
                          com.microsoft.jdbc.sqlserver.SQLServerDriver
hibernate.connection.url
     jdbc:microsoft:sqlserver://HOSTNAME:1433;DatabaseName=PersonPetsDB
hibernate.dialect = net.sf.hibernate.dialect.SybaseDialect
hibernate.show_sql = true
```

We will get back to executing this example shortly.

What does Hibernate offer?

Hibernate is an open source (released under LPGL) product for providing seamless persistence for Java objects. It uses reflections and yet provides excellent performance. It has support for over 30 different dialects (like drivers for different databases). It provides a rich query language to access objects, provide caching and JMX support. The is intended to provide high performance transparent persistence with low resource contention and small foot print.

JDO uses byte code enhancement while Hibernate uses runtime reflection to determine persistent properties of classes. A mapping property or configuration file is used to generate database schema and provide persistence. Figure below shows the mapping mechanism used by Hibernate:



A SessionFactory creates a Session object for transaction in a single threaded interaction. The Session acts as an agent between the application and the data store. This is the object

you will interact with to create, update and load objects. A Query class allows us to manage our queries and allows for parameterized queries as well.

Using Hibernate

Let us proceed with the example we started with. How do we create objects of Person and Dog and make it persistent? The CreatePerson.java code shown below does just that:

```
package com.agiledeveloper;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class CreatePerson
{
      public static void main(String[] args)
      {
            try
            {
                  Configuration config = new Configuration()
                        .addClass(com.agiledeveloper.Person.class)
                        .addClass(com.agiledeveloper.Dog.class);

                  SessionFactory sessionFactory
                                    = config.buildSessionFactory();
                  Session session = sessionFactory.openSession();
                  Transaction txn = session.beginTransaction();

                  Person john = new Person();
                  john.setFirstName("John");
                  john.setLastName("Smith");

                  Dog rover = new Dog();
                  rover.setName("Rover");
                  john.addPet(rover);
                  rover.setFriend(john);

                  session.save(john);
                  txn.commit();
                  session.close();
            }
            catch(Exception ex)
            {
                  System.out.println("Error: " + ex);
            }
      }
}
```

A Configuration object is first created. This object is used to create the SessionFactory. Each of the persistent types is introduced to the configuration object. The appropriate mapping files (*classname*.hbm.xml) will be consulted at runtime. The SessionFactory is used to open a Session, which in turn is used to start a transaction. Note how the objects of Person and Dog are created without any regard to persistence. The objects of Person (john) is finally made persistent by calling the save method on the Session object. Note

that the object of Dog (rover) will automatically be saved due to the cascade effect of update on the Person object. The transaction is committed and the session is closed at the end.

Let's now look at an example of loading the object created above:

```
package com.agiledeveloper;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class LoadPerson
{
      public static void main(String[] args)
      {
            try
            {
                  Configuration config = new Configuration()
                        .addClass(com.agiledeveloper.Person.class)
                        .addClass(com.agiledeveloper.Dog.class);

                  SessionFactory sessionFactory
                                    = config.buildSessionFactory();
                  Session session = sessionFactory.openSession();
                  Transaction txn = session.beginTransaction();

                  Person john = new Person();
                  session.load(john, new Long(3));

                  System.out.println("********** "
                        + john.getFirstName() + " "
                        + john.getLastName());

                  java.util.Iterator iter = john.getPets().iterator();

                  while(iter.hasNext())
                  {
                        Dog dog = (Dog) iter.next();
                        System.out.println("********** "
                                    + dog.getName());
                  }
                  txn.commit();
                  session.close();
            }
            catch(Exception ex)
            {
                  System.out.println("Error: " + ex);
            }
      }
}
```

In the above example, the load method on the Session object is called to fetch the object of Person. Note how a Person object to be loaded is created and sent to the load method along with the value of the primary key of the Person object. When the load returns the

Person object, the Dog object(s) related to the Person have not been loaded. This is due to the lazy (= true) attribute set in the mapping file. The Dog object is actually loaded on demands. This of course requires that the session be open until the Dog is fetched. One may easily test this by calling session.close() right after the session.load()statement. If the lazy load is set to false, there will not be any error. However, if lazy load is set to true, an exception is thrown.

The following code excerpt shows how to find a person (in FindPerson.java):

```
String query = "from Person p where p.lastName = 'Smith'";
java.util.List persons = session.find(query);

for(int i = 0; i < persons.size(); i++)
{
      Person aPerson = (Person)(persons.get(i));
      System.out.println("********** " + aPerson.getFirstName() + " " +
                              aPerson.getLastName());

      java.util.Iterator iter = aPerson.getPets().iterator();

      while(iter.hasNext())
      {
            Dog dog = (Dog) iter.next();
            System.out.println("--------------- " + dog.getName());
      }
}
```

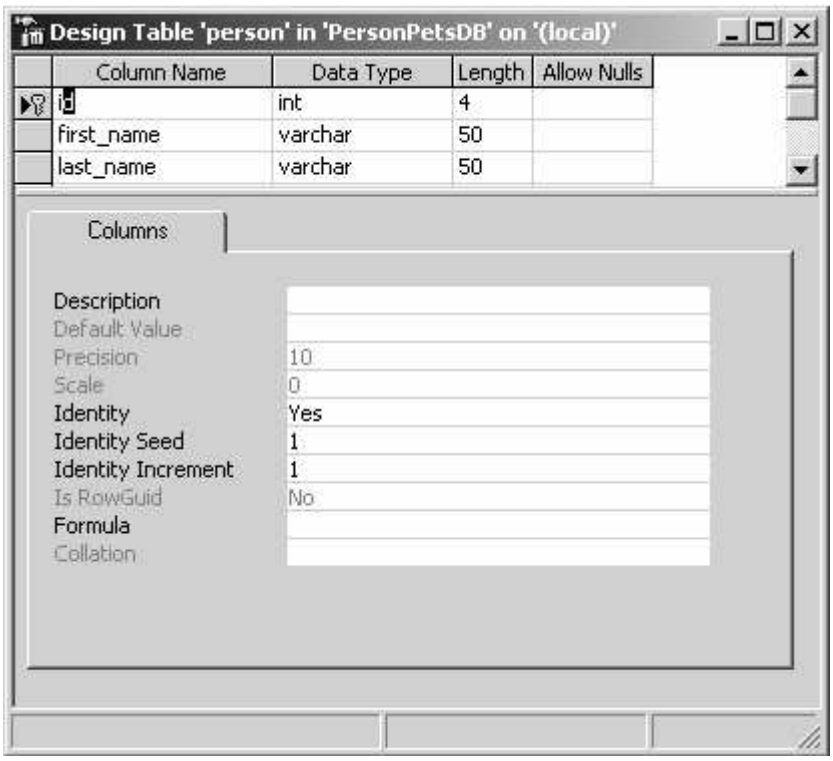
Here all persons with “Smith” as the last name are fetched from the database. The Hibernate Query Language (HQL) is useful to specify the query based on properties. Note how, in the above query, the “lastName” property is specified on an object reference “p” of Person. The query is based on fields of object instead of columns in the database.

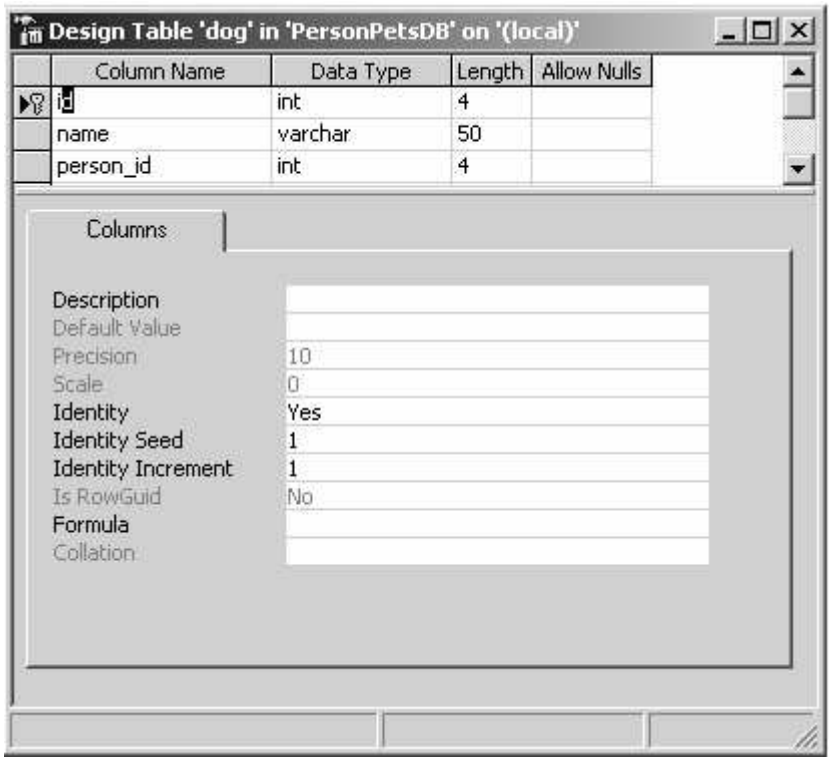
Pros and Cons of Hibernate

Hibernate provides highly efficient transparent persistence for Java objects. It leaves behind a very small footprint; has low resource requirements. It provides for high concurrency. Caching and lazy loading can further improve performance. Also, instead of saving an entire object, it updates only modified fields of an object. This leads to more optimized and efficient code than coding with JDBC. Further given a mapping property file, automatic code generation and schema generation tools are also available. On the downside, mapping an object to multiple tables is harder to realize.

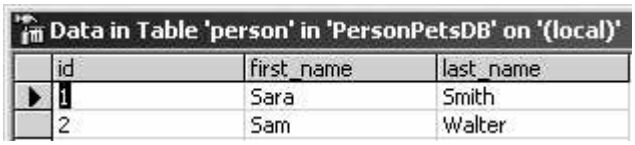
Running the Example

We had to download the SQL Server JDBC driver from microsoft’s web site. The related jar files (three in all) were copied to the Hibernate’s lib directory. A database named PersonPetsDB was created with two tables as shown below:



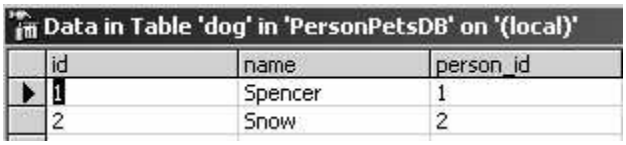


Some initial values were entered into these two tables as shown below:



Data in Table 'person' in 'PersonPetsDB' on '(local)'

	id	first_name	last_name
▶	1	Sara	Smith
	2	Sam	Walter



Data in Table 'dog' in 'PersonPetsDB' on '(local)'

	id	name	person_id
▶	1	Spencer	1
	2	Snow	2

The following Ant build.xml script was used to compile and run the program:

```
<project name="Hibernate" default="run">

   <property name="HBM-HOME" location="/programs/hibernate-2.1" />
   <property name="build.dir" value="output" />
   <property name="build.classes.dir" value="${build.dir}/classes"/>
   <property name="src.dir" value="src"/>
   <property name="cnfg.dir" value="cnfg"/>

   <path id="classpath.base">
      <pathelement location="${build.classes.dir}" />
      <pathelement location="${HBM-HOME}/hibernate2.jar" />
      <fileset dir="${HBM-HOME}/lib" includes="**/*.jar" />
   </path>

   <target name="prepare">
      <mkdir dir="${build.dir}" />
   </target>

    <target name="clean">
        <delete dir="${build.dir}"/>
    </target>

    <target name="compile" depends="clean">
      <mkdir dir="${build.classes.dir}"/>
      <javac srcdir="src"
           destdir="${build.classes.dir}"
           debug="on"
           deprecation="on"
           optimize="off">
           <classpath refid="classpath.base" />
      </javac>
      <copy file="${cnfg.dir}/Person.hbm.xml"
todir="${build.classes.dir}/com/agiledeveloper"
         overwrite="true" />
      <copy file="${cnfg.dir}/Dog.hbm.xml"
todir="${build.classes.dir}/com/agiledeveloper"
         overwrite="true" />
      <copy file="${cnfg.dir}/hibernate.properties"
todir="${build.classes.dir}"
         overwrite="true" />
    </target>

    <target name="runCreatePerson" depends="compile">
```

```
            <java classname="com.agiledeveloper.CreatePerson"
fork="true" dir="${build.classes.dir}">
                  <classpath refid="classpath.base" />
            </java>
    </target>

    <target name="runLoadPerson" depends="compile">
            <java classname="com.agiledeveloper.LoadPerson" fork="true"
dir="${build.classes.dir}">
                  <classpath refid="classpath.base" />
            </java>
    </target>

    <target name="runFindPerson" depends="compile">
            <java classname="com.agiledeveloper.FindPerson" fork="true"
dir="${build.classes.dir}">
                  <classpath refid="classpath.base" />
            </java>
    </target>

    <target name="run" depends="runCreatePerson">
    </target>
</project>
```

Running ant, compiled the code and ran the CreatePerson class. This added a Person (John Smith) to the person table and his pet (rover) to the dog table.

Running ant runLoadPerson produced the following result:

```
[java] Hibernate: select person0_.id as id0_, person0_.first_name as
first_name0_, person0_.last_name as last_name0_ from person person0_
where person0_.id=?
[java] ********** John Smith
[java] Hibernate: select pets0_.id as id__, pets0_.person_id as
person_id__, pets0_.id as id0_, pets0_.name as name0_, pets0_.person_id
as person_id0_ from dog pets0_ where pets0_.person_id=?
[java] ********** Rover
```

The output shows the SQL generated as well. This is due to the `hibernate.show_sql = true` in the hibernate.properties file. Setting this property to false will suppress these messages.

Moving the txn.commit() and session.close() statements to right after the session.load() call results in the following error:

```
[java] Hibernate: select person0_.id as id0_, person0_.first_name as
first_name0_, person0_.last_name as last_name0_ from person person0_
where person0_.id
=?
[java] ********** John Smith
[java] Error: net.sf.hibernate.LazyInitializationException: Failed to
lazily initialize a collection - no session or session was closed
```

However, if we modify the lazy attribute in person.hbm.xml to false, we get the following result:

```
[java] Hibernate: select person0_.id as id0_, person0_.first_name as
first_name0_, person0_.last_name as last_name0_ from person person0_
where person0_.id
=?
[java] Hibernate: select pets0_.id as id__, pets0_.person_id as
person_id__, pets0_.id as id0_, pets0_.name as name0_, pets0_.person_id
as person_id0_ from dog pets0_ where pets0_.person_id=?
[java] ********** John Smith
[java] ********** Rover
```

Note the difference between this output and the earlier output with lazy = true and txn.commit() and session.close() calls at the end. The lazy loading happens on the call to getPets() on the Person object.

Now, we run ant runFindPerson. The output we get is shown below:

```
[java] Hibernate: select person0_.id as id, person0_.first_name as
first_name, person0_.last_name as last_name from person person0_ where
(person0_.last_name='Smith' )
[java] ********** Sara Smith
[java] Hibernate: select pets0_.id as id__, pets0_.person_id as
person_id__, pets0_.id as id0_, pets0_.name as name0_, pets0_.person_id
as person_id0_ from dog pets0_ where pets0_.person_id=?
[java] ---------- Spencer
[java] ********** John Smith
[java] Hibernate: select pets0_.id as id__, pets0_.person_id as
person_id__, pets0_.id as id0_, pets0_.name as name0_, pets0_.person_id
as person_id0_ from dog pets0_ where pets0_.person_id=?
[java] ---------- Rover
```

Conclusion

While JDBC provides encapsulation of your Java code from the variations of database management systems, it unfortunately does not isolate the code from SQL queries. Hibernate provides transparent object persistence using runtime reflection. The code is far more isolated from the persistence mapping as this is kept separate in a mapping file. The application code is a lot cleaner and simpler to develop. All this comes with higher performance and small footprint as well.

References

1. http://www.hibernate.org
2. http://www.sourceforge.net/projects/hibernate