Article
Home> Articles > Java > Database

Hibernate Tutorial

This is a Hibernate tutorial to help you learn Hibernate and understand how it is used. By reading this Hibernate tutorial you will learn the main concepts, connect to databases using Hibernate by following real source code examples in the tutorial. Our Hibernate tutorial includes some videos to make it easy to learn.

Introduction to Java Hibernate

Developing enterprise applications involves a lot of efforts to store the business objects in the database. Usually,70% of the efforts to develop enterprise applications are writing the code to maintain database connections and save the entities to database. Most of the enterprise applications are developed using Object Oriented techniques. The business logic inside the application flows in the form of objects instead of textual messages as it used to be in the days of structural programming where we had set of variable to perform operations or indicate the state of an entity. Today,Object Oriented technologies are used to solve the complex problems and maintain the business flow. Only saving the objects to the database is a stage where the objects are transformed back to text and stored to the database. To understand this problem,consider the following code snippets.

This is typically how the connection is obtained in Java language.

```
Class.forName(driverClass);
java.sql.Connection connection =
java.sql.DriverManager.getConnection(databaseUrl,databaseUser,databasePassword);
```

Suppose we have an object of User class,and we need to store this object to the database. Consider the following function:

```
private void saveUser(User user) throws SQLException
{
    PreparedStatement pstmt = connection.prepareStatement(insert into users
values(?,?,?));
    pstmt.setInt(1,user.getId());
    pstmt.setString(2,user.getName());
    pstmt.setString(3,user.getEmail());
    pstmt.setString(4,user.getAddress());
    pstmt.execute();
}
```

The problem with this code is that this is scattered everywhere in the application which affects the maintainability of the application. There are many problems with this approach like:

- Code is scattered at different un-manageable places.
- If your design changes after developing the application,it will be too expensive to identify the places where changes are required.
- Difficult to identify and fix the bugs
- Managing the database connections is a difficult task since the SQL code is scattered,so are the database connections.
- Managing the transactions is a complex task.

Due to these problems,bug fixing is an accepted stage of an application development process which makes the application too expensive even after the successful completion of development cycle.

It is highly needed to have a mechanism to isolate the database persistent code and write it so efficiently that when there is a change in the database design,it can be easily implemented with the belief that there is no un-identified place which may show a bug later.

Benefits of Relational Database Systems:

searching and sorting is fast
Work with large amounts of data
Work with groups of data
Joining, aggregating
Sharing across multiple users and multiple locations
Concurrency (Transactions)
Support for multiple applications
Consistent Integrity
Constraints at different levels
Transaction isolation

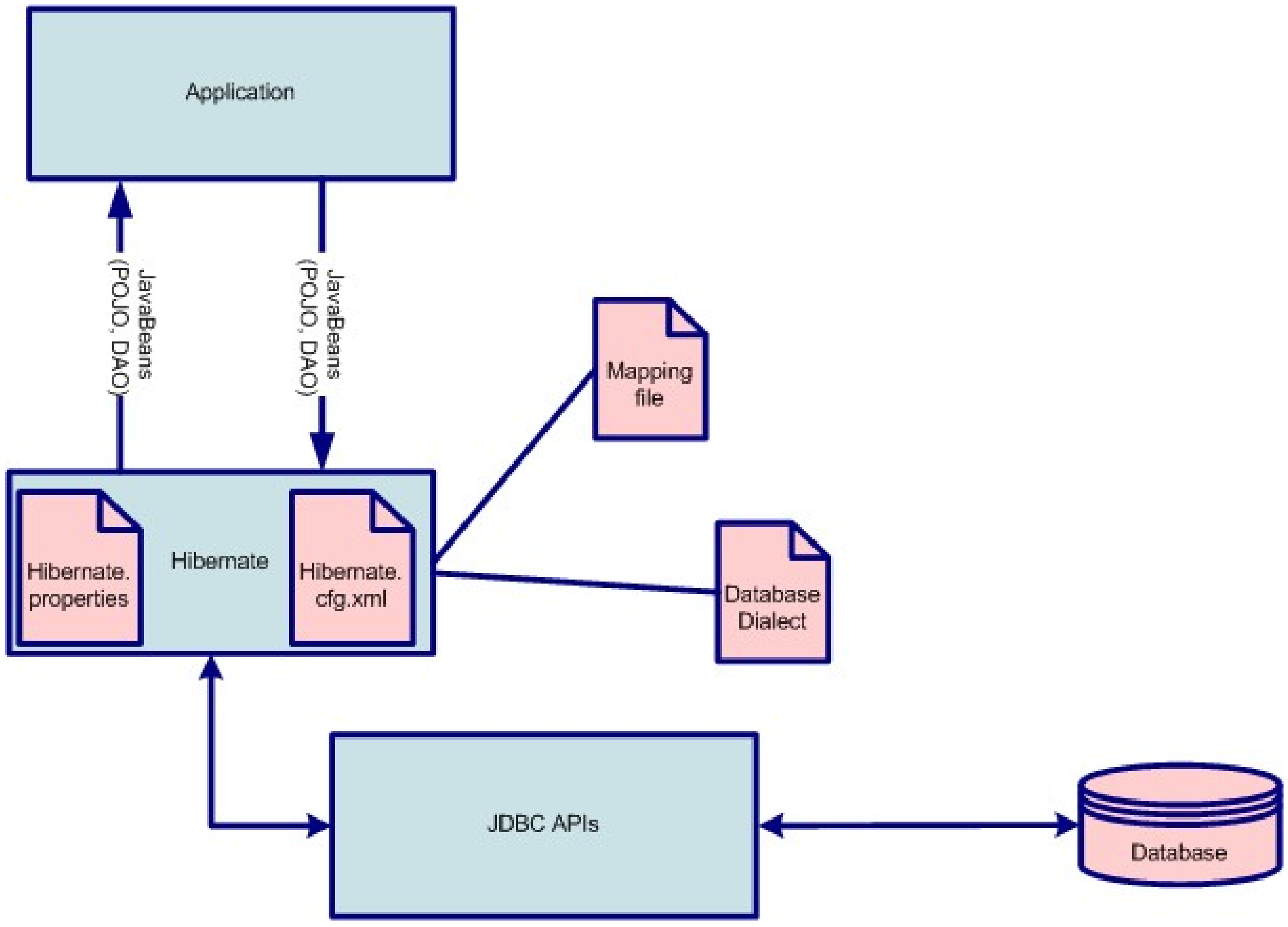
Issues with Relational Database systems:

Database Modeling does not support polymorphism
Business logic in the application server could be duplicated in the database as stored procedures
Does not make use of real world objects
Extra code required to map the Java objects to database objects.

Introduction to Hibernate

Hibernate is an open source object/relational mapping tool for Java that lets you develop persistent classes and persistent logic without caring how to handle the data. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types),but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's goal is to relieve the developer from 95% of common data persistence related programming tasks.

Hibernate makes it far easier to build robust,high-performance database applications with Java. You only need to write a simple POJO (Plain Old Java Object),create an XML mapping file that describes relationship between the database and the class attributes and call few Hibernate APIs to load/store the persistent objects.



The Object/Relational Mapping Problem

In object oriented systems,we represent entities as objects and classes and use database to persist those objects. Most of the data-driven applications today,are written using object oriented technologies. The idea of representing entities as set of classes is to re-use the classes and objects once written.

To understand the scenario,consider a web application that collects users personal information and stores them to the database. Also there could be a page that displays all the saved users. When implementing this kind of solution, we may store the data by getting the request parameters and simply putting it into SQL insert statements. Similarly when displaying,we can select the desired record from the database,and display them on dynamic web page by iterating through the result set. But wait...does it sound good to have persistent related code scattered everywhere on the web page? What if we need to modify the design of our database after we have developed few pages? Or even if we want to replace our low-cost development database with an enterprise-class production database that have entirely different SQL syntax? Do we need to change the SQL queries everywhere in our code?

To overcome this problem,we may want to develop JavaBeans representing the entities in the database. We can develop a Persistence Manager that can operate on the given JavaBean and perform several persistent operations like fetch,store,update,search etc. so that we don't need to write persistence related code in our core area of application. Advantage of using this technique is that we can pass on this simple JavaBean to any of the different layers of the application like a web layer,an EJB layer or any other layer we may have in our application.

We have isolated our persistence related code so that whenever there is a change in the design,or we change the database environment we can change our persistence code. Or we can have several copies of the Persistence Manager to work with different databases so that based on some configuration parameter,we can choose the right Persistence Manager instance. But this is still very difficult as we might ignore some database specific settings while writing so many Persistence Managers for different applications.

Here the concept of Object Relational mapping comes. By Object Relational mapping (O/R mapping) we mean the one-to-one mapping of class attributes with the columns of a database entity. Mapping includes the name of the attribute versus database field,the data type,and relation with other entities or classes. So if we write a framework that can take the object to relation mapping and provide persistence facilities on the given JavaBeans,we can use it for different applications and different databases. So,if anyone want to configure the application on a database that was not known at the time of writing the framework,he can simply write a mapping file and the application will start working on that environment.

Thanks to the open source community! we don't need to write such frameworks anymore. We have Hibernate which allows Object/Relation mapping and provides persistence to the abject with the help of simple APIs where we can specify the operations and the operands (JavaBeans).

JDBC

Java Database Connectivity (JDBC) is a set of APIs to connect to the database in a standard way. Since Hibernate provides Object/Relation mapping,it must communicate to the database of your choice. And that cannot happen without the use of a vendor dependent component that understands the target database and the underlying protocol. Usually the database vendors provide set of 'connectors' to connect with their database from external applications with different set of technologies / languages. JDBC driver is widely accepted connector that provides a fairly standard implementation of the JDBC specification. The point of discussion of JDBC here is,that you should know the basics of JDBC and how to use it with Hibernate.

Database connection can be provided by the Hibernate framework or by a JNDI Data source. A third method,user-provided JDBC connections,is also available,but it's rarely used. Typically,in standalone applications,we configure the JDBC connection properties in hibernte.cfg.xml file. In case of applications deployed in an application server,we may choose to use a container-managed data source,or provide a self-managed connection that we might already be using throughout the application. Hibernate itself does not impose a restriction to open or configure a separate connection for the Hibernate use only and keep other configurations to be used by rest of the application.

Before starting off with the Hibernate,make sure that you have essential set of JDBC libraries for the database you are going to use for your application.

The Hibernate Alternative

As discussed above,Hibernate provides data storage and retrieval by direct mapping of Object and Relation without letting the application developer worry about the persistence logic. There are few alternatives to Hibernate.

- You may choose to use plain SQL queries to store and retrieve data but that is not an alternate to Hibernate as using plain SQL queries does not provide maximum of the features provided by Hibernate.
- You can use Enterprise Java Beans. In particular,Entity Beans are good alternate to Hibernate. But again,you need to see if you really have the requirements of an EJB container?
- You can use the spring framework (www.springframework.org) to implement a DAO layer which would centralize as well as minimize any future need to change persistence implementations.
- There are many other Object/Relation mapping products available like OJB (ObJectRelationalBridge),Torque,Castor,Cayenne,TJDO etc. For more details see http://java-source.net/open-source/persistence

Hibernate Architecture and API

Hibernate is written in Java and is highly configurable through two types of configuration files. The first type of configuration file is named `hibernate.cfg.xml`. On startup,Hibernate consults this XML file for its operating properties,such as database connection string and password,database dialect,and mapping files locations. Hibernate searches for this file on the classpath. The second type of configuration file is a mapping description file (file extension `*.hbm`) that instructs Hibernate how to map data between a specific Java class and one or more database tables. Normally in a single application,you would have one hibernate.cfg.xml file,and multiple .hbm files depending upon the business entities you would like to persist.

Hibernate may be used by any Java application that requires moving data between Java application objects and database tables. Thus,it's very useful in developing two and three-tier J2EE applications. Integration of Hibernate into your application involves:

- Installing the Hibernate core and support JAR libraries into your project
- Creating a Hibernate.cfg.xml file to describe how to access your database
- Selecting appropriate SQL Dialect for the database.
- Creating individual mapping descriptor files for each persistable Java classes

Here are some commonly used classes of Hibernate.

1.5.1 SessionFactory (`org.hibernate.SessionFactory`)

SessionFactory is a thread safe (immutable) cache of compiled mappings for a single database. Usually an application has a single SessionFactory. Threads servicing client requests obtain Sessions from the factory. The behavior of a SessionFactory is controlled by properties supplied at configuration time.

1.5.2 Session (`org.hibernate.Session`)

A session is a connected object representing a conversation between the application and the database. It wraps a JDBC connection and can be used to maintain the transactions. It also holds a cache of persistent objects,used when navigating the objects or looking up objects by identifier.

1.5.3 Transaction (`org.hibernate.Transaction`)

Transactions are used to denote an atomic unit of work that can be committed (saved) or rolled back together. A transaction can be started by calling `session.beginTransaction()` which actually uses the transaction mechanism of underlying JDBC connection,JTA or CORBA.A Session might span several Transactions in some cases.

A complete list of Hibernate APIs can be found at http://www.hibernate.org/hib_docs/v3/api/.

Setting Up Hibernate

This section will explain how to get the Hibernate distribution and configure the application with hibernate.

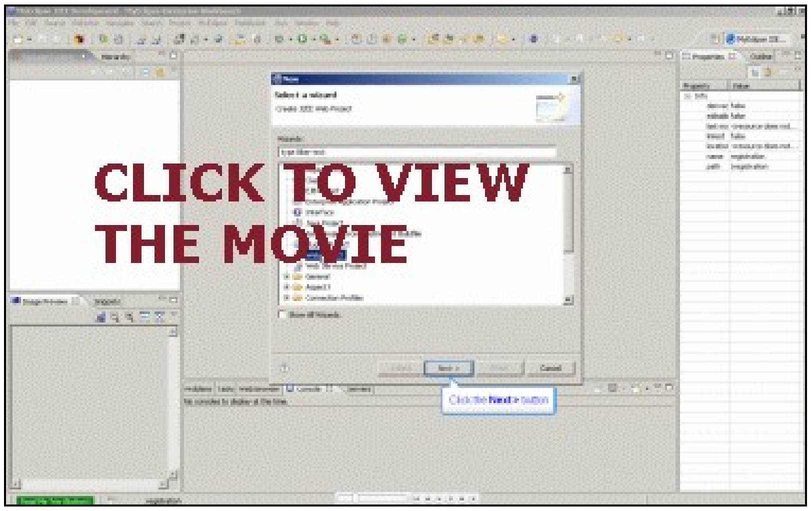
1. The Hibernate distribution is available at http://www.hibernate.org/6.html. Download and unzip the core package from this page.

2. Download and install JDK 1.4 (Sun or IBM) http://java.sun.com/j2se/downloads/index.html

3. Download and install Eclipse 3.2.1 from http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.2.1-200609210945/eclipse-SDK-3.2.1-win32 However,if you have any other version of Eclipse,that will work too.

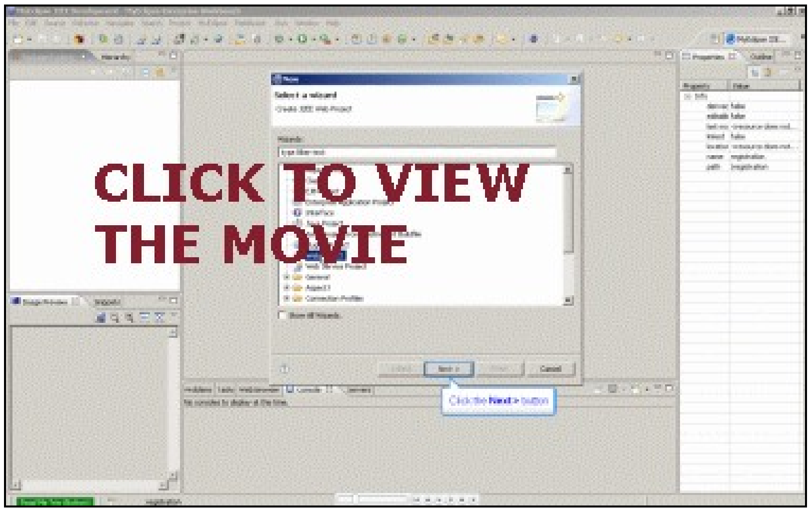
4. Download and Install Oracle or MySQL Database whichever is convenient to you. We will use Oracle Database 10g Express Edition. However we will let you know the appropriate steps for MySQL as well wherever necessary.

5. Run the Eclipse IDE and create a simple Java Project as shown below.



Setting up Hibernate - Add the Hibernate libraries

Add the Hibernate libraries to the project as shown below. Hibernate libraries can be found at the location where you downloaded and unzipped the Hibernate distribution.



Registration Case Study

To demonstrate the different steps involved in a Hibernate application,We will develop a console based application that can add,update,delete or search a user in the database. As discussed previously,we can use Oracle or MySQL database for storing the user information.

We’ll use a single table ‘users’ to store the user information with the following fields:

Column Name	Data Type	Constraints
USER_ID	NUMBER	PRIMARY KEY
FIRST_NAME	TEXT	NONE
LAST_NAME	TEXT	NONE
AGE	NUMBER	NONE
EMAIL	TEXT	NONE

Create this table in the database. The syntax for the oracle is given below.

```
CREATE TABLE USERS(
   USER_ID NUMBER PRIMARY KEY,
   FIRST_NAME VARCHAR(20),
   LAST_NAME VARCHAR(20),
   AGE NUMBER,
   EMAIL VARCHAR(40)
);
```

If you are using MySQL database,you can create the table using following syntax.

```
CREATE TABLE USERS(
   USER_ID NUMERIC PRIMARY KEY,
   FIRST_NAME CHAR(20),
   LAST_NAME CHAR(20),
   AGE NUMERIC,
   EMAIL CHAR(40)
);
```

Creating the Hibernate Configuration

Now it is the time to create our Hibernate configuration file. This file will contain the hibernate session configuration like the database driver,user name and password or a JNDI data source if you would like to use one,cache provider ad other session parameters. Also this file contains the entries for Object to Relation mapping that we will create later. Initially let's create the file with minimum required information and understand what does that information mean.

Create a new file `hibernate.cfg.xml` in `src` folder and add the following contents to this file.

```
<?xml version='1.0' encoding='utf-8'?>
  <!DOCTYPE hibernate-configuration PUBLIC   -//Hibernate/Hibernate Configuration DTD 3.0//EN
http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>
  <hibernate-configuration>
    <session-factory>
      <property name=connection.url>
        jdbc:oracle:thin:@localhost:1521:XE
      </property>
      <property name=connection.driver_class>
        oracle.jdbc.driver.OracleDriver
      </property>
      <property name=connection.username>
        SYSTEM
      </property>
      <property name=connection.password>
        manager
      </property>
      <!-- Set AutoCommit to true -->
      <property name=connection.autocommit>
        true
      </property>
      <!-- SQL Dialect to use. Dialects are database specific -->
      <property name=dialect>
        org.hibernate.dialect.OracleDialect
      </property>
      <!-- Mapping files -->
      <mapping resource=com/visualbuilder/hibernate/User.hbm.xml />
    </session-factory>
  </hibernate-configuration>
```

As you can see,the session factory configuration is set of few properties required for a database connection. **connection.url** is the JDBC URL required to connect to the database. **connection.driver_class** is the JDBC Driver class which we normally use in Class.forName while establishing a connection to the database. **connection.username** and **connection.password** are the database user name and password respectively. **connection.autocommit** sets the Autocommit property of the underlying connection. We have set it to true so that we don't need to commit the transactions ourselves unless we want to do it intentionally.

This configuration is for an Oracle database at my machine. If you are unaware of how to build a URL for an Oracle database,note that the you need to replace XE with an appropriate SID of your Oracle database,localhost with the IP address of the machine where Oracle is running if not on local machine,and 1521 with the port. In most cases,you will only change the SID and rest of the parameters will remain the same.

If you are using MySQL,you should put the appropriate URL,Driver class name,user name,password and the Dialect name. The Dialect for MySQL is `org.hibernate.dialect.MySQLDialect.` Driver class for the MySQL is `com.mysql.jdbc.Driver` and the URL for MySQL is jdbc:mysql://<server>:<port>/<database-name>. The default port for MySQL is 3306.

The next important step is to add the JDBC driver jar file to the project libraries just like we added the struts libraries. For oracle,it is ojdbc14.zip or ojdbc14.jar and can be found under Oracle installation/jdbc/lib. For example,if Oracle XE is installed in C:oraclexe,the ojdbc14.jar can be found on the following path.
C:oraclexeapporacleproduct10.2.0serverjdbclib. You can also download it from
http://www.minq.se/products/dbvis/drivers.html freely.You can download the MySQL JDBC driver freely from
http://dev.mysql.com/downloads/connector/j/5.0.htmll

Writing the first Java File

Let’s start with our first java class. Remember we have a table “users” in the database. Let’s write a simple bean that represents this table.

```
 package com.visualbuilder.hibernate;

 /**
  * @author VisualBuilder
  *
  */

public class User {
    private long userId = 0 ;
    private String firstName = ;
    private String lastName = ;
    private int age = 0;
    private String email = ;

    public int getAge() {
      return age;
    }
    public void setAge(int age) {
      this.age = age;
    }
    public String getEmail() {
      return email;
    }
    public void setEmail(String email) {
      this.email = email;
    }
    public String getFirstName() {
      return firstName;
    }
    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }
    public String getLastName() {
      return lastName;
    }
    public void setLastName(String lastName) {
      this.lastName = lastName;
    }
    public long getUserId() {
      return userId;
    }
    public void setUserId(long userId) {
      this.userId = userId;
    }
  }
```

Writing the mapping file

Mapping files are the heart of O/R mapping tools. These are the files that contain field to field mapping between the class attributes and the database columns.
Let's write a mapping file for the User class that we want to persist to the database.

```
<?xml version=1.0?>
  <!DOCTYPE hibernate-mapping PUBLIC
  -//Hibernate/Hibernate Mapping DTD 3.0//EN
  http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd >

  <hibernate-mapping>

      <class name=com.visualbuilder.hibernate.User table=USERS >
         <id name=userId type=java.lang.Long column=user_id >
             <generator class=increment />
         </id>
         <property name=firstName type=java.lang.String column=first_name length=20 />
         <property name=lastName type=java.lang.String column=last_name length=20 />
         <property name=age type=java.lang.Integer column=age length=-1 />
         <property name=email type=java.lang.String column=email length=40 />
      </class>

  </hibernate-mapping>
```

As you can see,we have mapped all of the attributes of the class User to the columns of the table we created previously. Note the `id` attribute which maps the `userId` field of class `User` is defined in a different way than the other properties. The `id` tag is used to indicate that this is a primary key,and a `<generator>` tag is used to generate the primary key using different techniques. We have used the `increment` class,but there are also available different classes to support different kind of key generation techniques like generating key from a sequence,selecting from database etc. We can also use a custom key generator class.

Let us now add the mapping file to the Hibernate configuration file hibernate.cfg.xml. After adding the mapping resource entry,the hibernate.cfg.xml looks like this.

```
<?xml version='1.0' encoding='utf-8'?>
  <!DOCTYPE hibernate-configuration PUBLIC
    -//Hibernate/Hibernate Configuration DTD 3.0//EN http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>
  <hibernate-configuration>
    <session-factory>

      <property name=connection.url>jdbc:oracle:thin:@localhost:1521:XE</property>
      <property name=connection.driver_class>oracle.jdbc.driver.OracleDriver</property>
      <property name=connection.username>SYSTEM</property>
      <property name=connection.password>manager</property>

      <!-- Set AutoCommit to true -->
      <property name=connection.autocommit>true</property>

      <!-- SQL Dialect to use. Dialects are database specific -->
    <property name=dialect>net.sf.hibernate.dialect.OracleDialect</property>

    <!-- Mapping files -->
   <mapping resource=com/visualbuilder/hibernate/User.hbm.xml />

  </session-factory>
</hibernate-configuration>
```

Writing the Business Component

So far,we have written a User class that we want to persist and a mapping file that describes the relationship of each of the attributes of the class to the database columns. We have also configured the Hibernate SessionFactory (hibernate.cfg.xml) to use the User.hbm.xml. Now it is the time to write a business component that can perform different operations on the User object.

7.1 Session

Before writing the business component,we need to understand a hibernate '`session`'. As discussed in section 1.5,a hibernate session is an instance of `org.hibernate.SessionopenSession()` of `SessionFactory`. Every database operation begins and ends with a `Session`. For example,we can `beginTransaction()`,`save()`,`update()` or `delete()` an object with the help of a `Session`. and is obtained by calling

Our business component will use the Session to perform its business operations.

```
package com.visualbuilder.hibernate.client; import org.hibernate.Session; import com.vis
```

Writing the Test Client

Let us write a test client that utilizes the different methods of the UserManager we have just introduced. We will obtain the `SessionFactory` and open a session and then call different methods of the `UserManager` to see if they work or not.

```
package com.visualbuilder.hibernate.client; import org.hibernate.*; import org.hibernate
{   SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory

public User testSaveUser(UserManager manager)  {       User user = buildUser(); manager.
      return user;
  }

  public void testUpdateUser(UserManager manager,User user)  { user.setFirstName(Andrew

  public void testDeleteUser(UserManager manager,User user)  { manager.deleteUser(user)

    User user = client.testSaveUser(manager);

    client.testUpdateUser(manager,user);

    client.testDeleteUser(manager,user);
session.flush(); } }
```

Managing Associations

Association is a relationship of one class to another class known as 'relationships' in database terminology. Usually the database design involves the master detail tables to represent the relationship of one entity to another. Struts provides a mechanism to manage one-to-many and many-to-many relationships. Before moving forward,let's create the necessary database tables to show associations.

Let's create a table `phone_numbers` to represent phone numbers of a user. Following script can be used to create the table in Oracle.

```
CREATE TABLE PHONE_NUMBERS
(
    USER_ID NUMBER REFERENCES USERS(USER_ID),
    NUMBER_TYPE VARCHAR(50),
    PHONE NUMBER,
    PRIMARY KEY(USER_ID,NUMBER_TYPE)
);
```

The same table can be created in MySQL using the script given below.

```
CREATE TABLE PHONE_NUMBERS
(
    USER_ID NUMERIC NOT NULL REFERENCES USERS(USER_ID),
    NUMBER_TYPE CHAR(50) NOT NULL,
    PHONE NUMERIC,
    PRIMARY KEY(USER_ID,NUMBER_TYPE)
);
```

We have two tables in the database: USERS and the PHONE_NUMBERS with one to many relationship such that one User many contain 0 or more phone numbers. Create a class PhoneNumber and create its mapping file following the steps to create User class. The code for the PhoneNumber class is shown below.

```
package com.visualbuilder.hibernate;
import java.io.Serializable;
/**
 * PhoneNumber
 * @author VisualBuilder
 */
public class PhoneNumber implements Serializable
{
private long userId = 0;
private String numberType = home;
private long phone = 0;
public String getNumberType() {
return numberType;
}
public void setNumberType(String numberType) {
this.numberType = numberType;
}
public long getPhone() {
```

```
return phone;
}
public void setPhone(long phone) {
this.phone = phone;
}
public long getUserId() {
return userId;
}
public void setUserId(long userId) {
this.userId = userId;
}
}
```

Note that we have made the PhoneNumber class Serializeable. This is because we need a composite key in this class,and Hibernate requires tat the class that represents the composite key must be Serializeable. The mapping file 'PhoneNumber.hbm.xml' contains following text.

```
<?xml version=1.0?>
<!DOCTYPE hibernate-mapping PUBLIC
   -//Hibernate/Hibernate Mapping DTD 3.0//EN
   http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd >

<hibernate-mapping>

 <class name=com.visualbuilder.hibernate.PhoneNumber table=PHONE_NUMBERS >
    <composite-id>
        <key-property column=USER_ID  name=userId type=java.lang.Long />
        <key-property column=NUMBER_TYPE  name=numberType type=java.lang.String/>
     </composite-id>

     <property name=phone type=java.lang.Long>
        <column name=PHONE precision=22 scale=0 />
     </property>

 </class>
</hibernate-mapping>
```

Add the mapping resource for PhoneNumber in hibernate.cfg.xml file. To do this,locate the line <mapping resource=com/visualbuilder/hibernate/User.hbm.xml /> in the hibernate.cfg.xml file and add the following line just after this line.

```
<mapping resource=com/visualbuilder/hibernate/PhoneNumber.hbm.xml />
```

We want to write few lines of code so that when we select a user,we automatically get all the phone numbers associated with this user. To do this,add a method 'getPhoneNumbers' that returns a list of users in class User and 'setPhoneNumbers' that takes a List as argument. The complete listing of class User after adding these two methods is shown below.

```
package com.visualbuilder.hibernate;
/**
 * @author VisualBuilder
 *
 */
public class User {
private long userId = 0;
private String firstName = ;
private String lastName = ;
private int age = 0;
private String email = ;
private java.util.Set phoneNumbers = new java.util.HashSet();
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
public String getFirstName() {
return firstName;
}
public void setFirstName(String firstName) {
this.firstName = firstName;
}
public String getLastName() {
return lastName;
}
public void setLastName(String lastName) {
this.lastName = lastName;
}
public long getUserId() {
return userId;
}
public void setUserId(long userId) {
this.userId = userId;
}
public java.util.Set getPhoneNumbers() {
return phoneNumbers;
}
public void setPhoneNumbers(java.util.Set phoneNumbers) {
if(phoneNumbers != null)
this.phoneNumbers = phoneNumbers;
}
}
```

The phoneNumbers HashTable will be used to store all phone numbers associated with this User. The new changes are highlighted in gray. We have added the methods for getting and setting the phone umbers,we need to add the mapping for the phone numbers now. Add the following block in User.hbm.xml before the </class> tag.

```
<set name=phoneNumbers cascade=all>
    <key column=USER_ID/>
    <one-to-many class=com.visualbuilder.hibernate.PhoneNumber />
</set>
```

As you can see,the tags are very simple. The cascade attribute of set tells the Hibernate how to deal with the child records when a parent is deleted or a child is added.

Now it is the time to test our functionality. Remember the TestClient class where we had few blocks of code explaining the save,update and delete functionality. For the time being,comment the call to `testDeleteUser` and replace the contents of method `testUpdateUser` with the following code.

```
PhoneNumber ph = new PhoneNumber();
ph.setUserId(user.getUserId());
ph.setNumberType(Office);
ph.setPhone(934757);
user.getPhoneNumbers().add(ph);

ph = new PhoneNumber();
ph.setUserId(user.getUserId());
ph.setNumberType(Home);
ph.setPhone(934757);
user.getPhoneNumbers().add(ph);

manager.saveUser(user);
System.out.println(User updated with ID = user.getUserId());
```

We are adding two phone numbers to the recently saved user objects. Note that we even don't need to save the PhoneNumber. These are automatically saved as we add them to the User. Query the database and see how these instances are persisted. Once you see the values in the database,re-enable the code to delete the User object and see what happens with the child PhoneNumber objects.

So far,we have dealt with one to many association,we can configure the Hibernate for one to one and many to many associations. For more details on how to make Hibernate configuration for other kind of associations,see http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#tutorial-associations

Finding by primary key

In the previous section,we tested the basic functionality of our hibernate application. With few lines of code,we were able to perform the basic operations like insert,update and delete on the database table,and add few children without writing a single SQL query. In enterprise applications,an efficient search mechanism is highly needed. Hibernate provides a set of different techniques to search a persisted object. Let us see a simple mechanism to find an Object by primary key without using a query.

Add the following method in UserManager class.

```
public User getUser(long userId)
{
User user = (User)session.get(User.class, new Long(userId));
return user;
}
```

This is the whole logic of finding a user by primary key. Let's test this code by adding the following method in TestClient.

```
  public void testFindByPk(UserManager manager)
{
User user = manager.getUser(1);//Find the user with id=1
if(user == null) {
System.out.println(No user found with ID=1);
}else {
System.out.println(User found with ID= user.getUserId() n
tName= user.getLastName() n
tEmail= user.getEmail()
);
java.util.Iterator numbers = user.getPhoneNumbers().iterator();
while(numbers.hasNext()) {
PhoneNumber ph = (PhoneNumber)numbers.next();
System.out.println(ttNumber Type: ph.getNumberType() n
ttPhone Number: ph.getPhone());
}
}
}
```

Add the call to this method in main. To do this,add the following line in `main`.

```
client.testFindByPk(manager);
```

Replace the user id 1 by some reasonable user id that exists in the database and there are few phone numbers added against that user id. You will see the following output.

```
User found with ID=1
Name=Elison
Email=john@visualbuilder.com
Number Type:Office
Phone Number:934757
Number Type:Home
Phone Number:934757
```

Hibernate Query Language (HQL)

In the previous section,we tried to find a persisted user by user id which was primary key in that case. What if we want to search a user by user name,age or email and none of these attributes is a primary key. Hibernate provides a query language similar to the standard SQL to perform operations on the Hibernate objects. The advantage of using HQL instead of standard SQL is that SQL varies as different databases are adopted to implement different solutions,but HQL remain the same as long as you are within the domain of Hibernate; no matter which database you are using. Let us learn to embed HQL in our example to find the users with age greater than 30. Before doing this,make sure that you have some of the users over the age of 30. For example,i issued the following query on my Oracle console to update the age of few users.

update users set age=31 where user_id in(5,8,9,11,13,14);

Modify this query to include some of the records in your users table.

Let's write the logic to find the users in our business component. Add the following method in UserManager.

```
public java.util.List getUsersByAge(int minAage) { org.hibernate.Query q = session.creat
```

Note that the query uses User which is the object name and not the table name which is Users. Similarly the u.age is an attribute of the User object and not the column name of the table. So whatever the name of the table may be,and whatever the presentation of that table or columns at the database level may be,we will use the same standard syntax while communicating with the hibernate. Also note the parameter minAge. The parameters in an HQL query are represented with :(colon) character and can be bound using the index or the parameter name in org.hibernate.Query.

Let's write a piece of ode to test this functionality. Add the following method in TestClient.

```
public void testFindByAge(UserManager manager) { java.util.Iterator users = manager.getU
```

Add the call to this method in main. To do this,add the following line in `main`.
`client.testFindByAge(manager);`

Run the program and see that all the users we modified above are displayed with respective phone numbers.

Following points should be kept in mind when working with HQL.

- Queries are case insensitive,except the Java class and attribute names. Hence SELECT and select are the same,but User and user are not.
- If the class or package does not find a place in imports,then the objects have to be called with the package name. For example,if com.visualbuilder.hibernate.User is not imported,then the query would be from com.visualbuilder.hibernate.User and so on.

For more details on HQL,visit http://www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html

Using native SQL

In previous section,we learnt to use the Hibernate Query Language (HQL) that focused on the business entities instead of the vendor dependent syntax. This does not mean that we are bound to use the HQL throughout the Hibernate application if we want to perform some database operations. You may express a query in SQL,using `createSQLQuery()` and let Hibernate take care of the mapping from result sets to objects. Note that you may at any time call `session.connection()` and use the JDBC `Connection` directly. If you chose to use the Hibernate API,you must enclose SQL aliases in braces.

Let's see how to use the SQL queries by adding the functionality to find a user by user id using native SQL in our UserManager. Add the following method in UserManager class.

```
public User getUserById(long userId)
{
   org.hibernate.SQLQuery query = session.createSQLQuery(
      SELECT u.user_id as {u.userId},u.first_name as {u.firstName},u.last_name as {u.lastName},u.age as
{u.age},u.email as {u.email}
      FROM USERS {u} WHERE  {u}.user_id= userId );
   query.addEntity(u,User.class);
   java.util.List l = query.list();
   java.util.Iterator users = l.iterator();
   User user = null;
   if(users.hasNext())
      user = (User)users.next();
   return user;
}
```

In the above code,the result type is registered using `query.addEntity(u,User.class)` so that Hibernate knows how to translate the ResultSet obtained by executing the query. Also note that the aliases in the query are placed in braces and use the same prefix u as registered in query.addEntity(...). This way Hibernate knows how to set attributes of the generated object. Also this way we can eliminate the confusion of attributes when more than one tables are used in query and both of them contain the same column name. `query.list()` actually executes the query and returns the ResulSet in the form of list of objects. We knew that this query is going to return only one object,so we returned only the first object if available.

Let's test this code by adding the following method in the TestClient.

```
public void testFindByNativeSQL(UserManager manager)
{
User user = manager.getUserById(2);
System.out.println(User found using native sql with ID= user.getUserId() n
tName= user.getLastName() n
tEmail= user.getEmail()
);
java.util.Iterator numbers = user.getPhoneNumbers().iterator();
while(numbers.hasNext()) {
PhoneNumber phone = (PhoneNumber)numbers.next();
System.out.println(ttNumber Type: phone.getNumberType() n
ttPhone Number: phone.getPhone());
}
}
```

Add the call to this method in main. To do this,add the following line in `main`.

```
client.testFindByNativeSQL(manager);
```

Be sure to pass a valid user id to this method that exists in the database with few valid phone numbers. This code will result in the following output.

```
User found using native sql with ID=2
Name=Elison
Email=john@visualbuilder.com
Number Type:Office
Phone Number:934757
Number Type:Home
Phone Number:934757
```

For more details on using native SQL in Hibernate,see Hibernate documentation.

Using Criteria Queries

So far we have seen how to use HQL and native SQL to perform certain database operations. We saw that the HQL was much simpler than the native SQL as we had to provide database columns and the mapping attributes along with mapping classes too to get fully translated java objects in native SQL. Hibernate provides much simpler API called Criteria Queries if our intention is only to filter the objects or narrow down the search. In Criteria Queries,we don't need to provide the select or from clause. Instead,we just need to provide the filtering criteria. For example name like '%a',or id in (1,2,3) etc. Like HQL,the Criteria API works on java objects instead on database entities as in case of native SQL.

Let's write a method in UserManager to filter the users that have user id within a set of passed user id's.

```
public java.util.List getUserByCriteria(Long[] items)
{
org.hibernate.Criteria criteria = session.createCriteria(User.class);
criteria.add(org.hibernate.criterion.Restrictions.in(userId, items));
return criteria.list();
}
```

This method returns the users and associated phone numbers that have one of the user id in items passed as an argument. See how a certain filter criteria is entered using `Restrictions` class. Similarly we can use `Restrictions.like`,`Restrictions.between`,`Restrictions.isNotNull`,`Restrictions.isNull` and other methods available in `Restrictions` class.

Let's write the test code to verify that the filter we provided using Criteria works. Add the following method in TestClient.

```
public void testFindByCriteria(UserManager manager)
{
java.util.Iterator users = manager.getUserByCriteria(new Long[]{new Long(1),new Long(2)
while(users.hasNext())
{
User user = (User)users.next();
System.out.println(User found with ID=+user.getUserId()+n +
tName=+user.getLastName()+n +
tEmail=+user.getEmail() +
);
java.util.Iterator numbers = user.getPhoneNumbers().iterator();
while(numbers.hasNext()) {
PhoneNumber phone = (PhoneNumber)numbers.next();
System.out.println(ttNumber Type:+phone.getNumberType()+n +
ttPhone Number:+phone.getPhone());
}
}
}
```

Add the call to this method in main. To do this,add the following line in `main`.
client.testFindByCriteria(manager);

We passed the two user ids to the function. So we should get the two users (if we have in the database with these user id's) with appropriate phone number entries if they exist in the database. Here is what gets displayed under my environment. You should adjust the appropriate user id's to get the results.

```
User found with ID=1
```

```
Name=Elison
Email=john@visualbuilder.com
Number Type:Office
Phone Number:934757
Number Type:Home
Phone Number:934757
User found with ID=2
Name=Elison
Email=john@visualbuilder.com
Number Type:Home
Phone Number:934757
Number Type:Office
Phone Number:934757
```

Using Ant to run the project

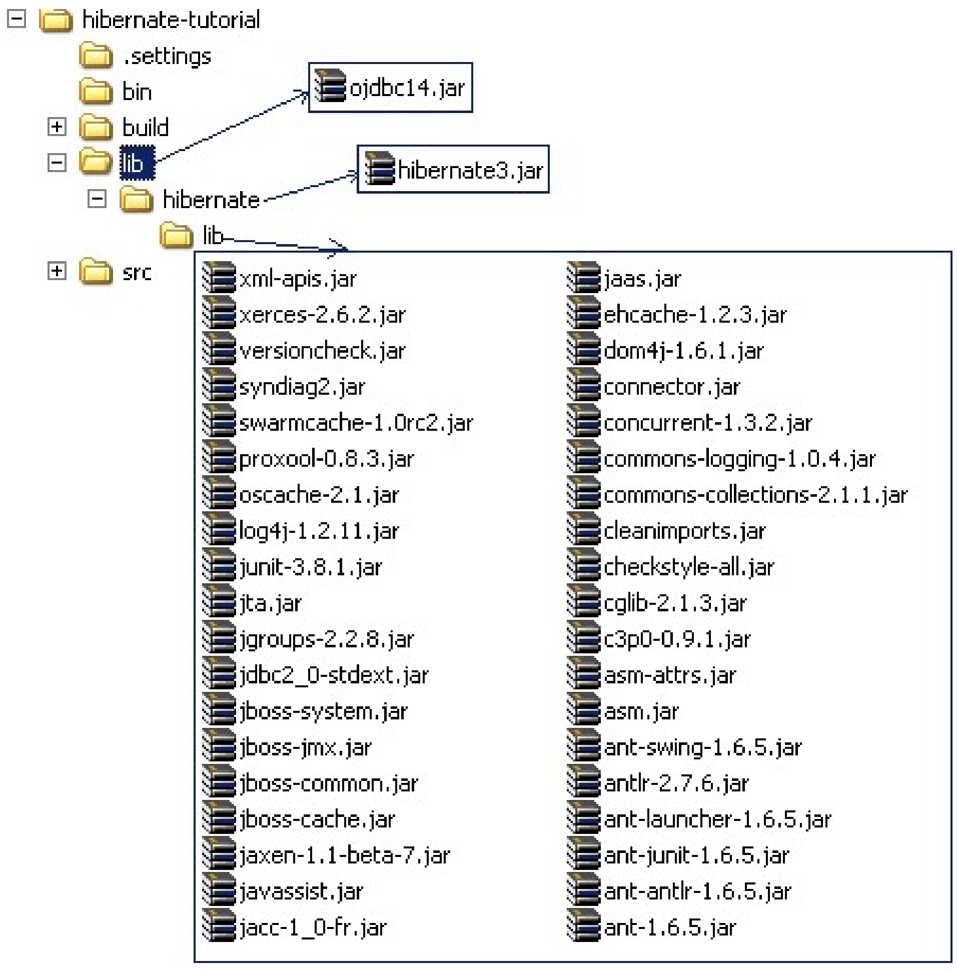
In previous sections,we have learnt the widely used components of Hibernate. Our development environment was eclipse and a set of Hibernate libraries. Usually ant is used to build and run the Hibernate applications. In this section,we will configure our project to use ant and build without any help of eclipse even from command line.

To start with ant:

- Download and install the latest available version Apache Ant from http://ant.apache.org
- Set environment variable JAVA_HOME to point to the available JDK location like d:javaj2sdk1.4.2_04
- Set environment variable ANT_HOME to point to the ant installation directory like d:javaapache-ant-1.6.2
- Add %ANT_HOME%bin to your PATH environment variable so that the ant binaries are accessible.

Our ant installation is ready. Let's configure our project to use ant.

Create a directory lib at the root of our project and place the libraries required by our project in this folder. The libraries include all the Hibernate jar files we added in eclipse project libraries and the JDBC driver. See the directory structure below to find where to place the libraries.



Create an xml file at the root of the project. i.e. where the src and lib resides; and add the following text in this file.

```
<?xml version=1.0?> <project name=hibernate-tutorial default=run basedir=.>     <!-- set
```

This is the main build script that can be used to automatically build the source,copy the configuration files(*.hbm.xml and *.cfg.xml),build the jar file and execute the program. Open the command prompt,change your current directory to the project root,and issue the following command:
ant
The out put of this command is shown below:

```
E:eclipseprojectshibernate-tutorial>ant Buildfile: build.xml init: [mkdir] Created dir:
```

So,you can see that the ant did a magic for us by doing all the configuration,including the necessary libraries and automatically executing the client program for us.

Using Middlegen to generate source

In previous section,we configured our application to use ant. So far,we have been writing he O/R mapping files (hbm) and writing java code for them manually. Frankly speaking,there is no such need to put so much effort for this kind of stuff as there are different tools available that can be used to generate mapping files from database and then java code from these files. Middlegen is one of these tools,and can be downloaded from http://sourceforge.net/project/showfiles.php?group_id=36044

We will use middlegen in our build script build.xml to generate source from database. Let's add the required libraries to our lib directory.

1. Download and unzip the middlegen distribution along with the dependencies.

2. Copy the jar files from middlegen distribution with dependencies to the lib directory. Also download the hibernate tools distribution from http://hibernate.org/6.html and place the jar file in lib directory. You can find the dependencies and the tools in middlegen/samples/lib directory.

3. Add the following properties in the properties section of the build.xml:

```
 <property name=hbmdir value=hbm />
<property name=db.url value=<database connection url> />
<property name=db.driver value=<database driver class> />
<property name=db.user value=<database user name> />
<property name=db.password value=<database user password> />
```

Replace the values of the respective properties with your database JDBC url,Driver name,user name and password. We will use these properties to generate source from source database. The hbmdir property will point to the location where we want to place hbm files.

4. Add the following line in init target:

```
 <mkdir dir=${hbmdir}/>
```

5. Add the following line in clean target so that the hbm directory is also deleted with other build directories.

```
 <delete quiet=true dir=${hbmdir}/>
```

6. Add the following target in build.xml:

```
<target name=gen-hbm depends=init>
  <taskdef name=middlegen classname=middlegen.MiddlegenTask classpathref=cp />
    <middlegen appname=hibernate-tutorial prefsdir=${hbmdir}
  gui=false databaseurl=${db.url}
  driver=${db.driver}
  username=${db.user}
  password=${db.password}
  schema=
  catalog=
  includeViews=false >
 <table generate=true name=USERS />
 <table generate=true name=PHONE_NUMBERS />
   <!-- Plugins -->
    <hibernate
      destination=${hbmdir}
      package=com.visualbuilder.hibernate
     javaTypeMapper=middlegen.plugins.hibernate.HibernateJavaTypeMapper
   />
 </middlegen>
 <taskdef name=hbm2java classname=net.sf.hibernate.tool.hbm2java.Hbm2JavaTask classpathref=cp />
 <hbm2java output=${build.gen-src.dir}>
    <fileset dir=${hbmdir}>
      <include name=**/*.hbm.xml/>
    </fileset>
 </hbm2java>
 </target>
```

Full contents of this file are given below:

```
<?xml version=1.0?> <project name=hibernate-tutorial default=run basedir=.>    <!-- set
```

Middlegen 3 is not available yet,and middlegen 2.1 does not support Hibernate 3. So,to generate java source from hbm using Hibernate Tools,

Replace the database properties values and run the following command on command prompt while residing in the project root directory:

ant gen-hbm

This will generate two hbm files in package folders under the hbm directory.

We have separately generated the hbm and source. If we just replace the hbm and the java files in package com.visualbuilder.hibernate,we will have few conflicts because middlegen has transformed the data types differently. We have used long for the numeric types,but the middlegen actually represents numeric as BigDecimal. So there will be few syntax errors in the client only; on removing which the code should work smoothly.

Thanks for reading the Java Hibernate Tutorial

What's Next?

Now that you have learnt how to create some simple Hibernate code and grasped the fundamentals of Hiernate programming,you can build on your knowledge and become an expert.

Make sure you visit our famous JSP tutorial (one of the best on the Internet)

Click here to see the JSP Tutorial

More Tutorials

Please view our *JSP and J2EE Design Tutorial* to learn more about how to use JSP.

Click here to start the JSP and J2EE Design Tutorial

Learn about the *JSP Struts Framework* for the next step of practically ising JSP.

JSP Struts Tutorial (New!)

Join our Java and JSP Group and we'll keep you up to date

Do you want to be kept up to date with the new JSP tutorial updates and the latest JSP code,articles and news?

Visit Java Group and click the Join button.

Visit JSP Group and click the Join button. You can start discussing with others learning this technology.

It's free and you can set how you want to your email updates.

Date entered : 13th May 2007

Rating :*No Rating*

Submitted by : visualbuilder