**Hibernate Simple Example**

**Note: This notes on Hibernate Example gives the basic steps of writing a Hibernate program. Explained in simple steps.**

Here, a **Student** record is inserted into database table using Hibernate. All the code is explained very clearly with the relevant **XML** files. The execution part of this program is given separately in [Simple Hibernate Program – Step-by-step Execution](http://way2java.com/hibernate/simple-hibernate-program-step-by-step-execution-with-screenshots/) with screenshots. It advised to read the Hibernate Tutorial before going into this program.

First let us go for a simple program where a **Student** object is inserted (persisted) into MS ACCESS database table **school**.

|  |  |  |  |
| --- | --- | --- | --- |
| STUDENT CLASS | | SCHOOL TABLE | |
| JAVA INSTANCE VARIABLE | TYPE | TABLE COLUMN NAME | TYPE |
| Sid | int | STUD\_ID (primary key) | number(4) |
| Sname | String | STUD\_NAME | VARCHAR2(15) |
| Smarks | double | STUD\_MARKS | number(4,1) |
| Sjoindate | Date | STUD\_JOIN-DATE | Date |

Now you are ready with school table and further require 4 programs.

1. **studentdetail.java – Java class written with JavaBean syntax.**
2. **hibernate.cfg.xml – A Configuration file**
3. **hibernate.hbm.xml**  **– A Mapping file.**
4. **StudentClient.java – Java Client program that inserts (or stores or persists) Student objects in database.**

Now let us see each of the above 4 files and discuss line-wise.

**1. Java program Student.java with JavaBean syntax with setter and getter methods (Persistent class)**

The variables of this Java program correspond to columns in a database table. We create an object **Student** class, set the properties and persist (storing) in database table **school**.

package com;

import java.util.\*;

/\*\*

\*

\* @author NCV

\*/

public class studentdetail {

private int sid;

private String sname;

private double smarks;

private Date sjoindate;

public studentdetail()

{

// TODO Auto-generated constructor stub

}

public int getSid() {

return sid;

}

public void setSid(int sid) { this.sid = sid; }

public String getSname() { return sname; }

public void setSname(String sname) { this.sname = sname; }

public double getSmarks() { return smarks; }

public void setSmarks(double smarks) { this.smarks = smarks; }

public Date getSjoindate() { return sjoindate; }

public void setSjoindate(Date sjoindate) { this.sjoindate = sjoindate; }

}

In JavaBean syntax, each property is represented by one instance variable. Every property should have one set and one get method. Here, four properties are declared as **sid**, **sname**,**smarks** and **sjoindate**. Values for these variables are given in client program. Values given are placed in the database table **schoo**l.

The **Student** class implements [Serializable](http://way2java.com/serialization/what-is-serialization/) interface for object persistence. The class should follow JavaBean syntax rules like having a default constructor and setter and getter methods.

I used purposefully different data types so that a learner gets acquainted of their usage in Hibernate.

Now you are ready with the class **studentdetail** and table **school**.

It requires two XML files – **hibernate.cfg.xml** and **hibernate.hbm.xml**.

**2. Configuration File – hibernate.cfg.xml**

This file comes with Hibernate software distribution. Just override as per your convenience. This XML file mainly tells Hibernate what database you are using to insert the data along with user name and password etc. The file should have an extension **.cfg.xml**

<?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="hibernate.connection.driver\_class">sun.jdbc.odbc.JdbcOdbcDriver</property>

<property name="hibernate.connection.url">jdbc:odbc:test</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<mapping resource="hibernate.hbm.xml"/>

</session-factory>

</hibernate-configuration>

The syntax is almost self-explanatory but for a few. **Dialect** tells the name of the database we are using. Hibernate supports Oracle. The other databases Hibernate supports are given at the end of this notes.

**<session-factory>** is responsible to connect to database. Programmer should provide all the required information in the form of XML tags. The first five **<property>**elements give the database specific information including the driver being used. The information of this XML file is used by **SessionFactory** object in the client program to connect to database. If the same Java bean values (called as properties) are to be inserted in multiple databases (like one table in Oracle and one table in MS-Access etc.), multiple **SessionFactory** objects are required.

**show\_sql** shows on the Console (of MyEclipse), the SQL statements created by **hibernate.hbm.xml** is the XML file where mappings between Java variables and database columns are given.

**3. Mapping File – hibernate.hbm.xml**

This XML file tells what instance variable of **studentdetail** class is to be mapped to what column of table **school**.

The file should have an extension**.hbm.xml**. The name is given as Student just to remember **hibernate.hbm.xml**file belongs to class Student, but for it can be any name.

<?xml version="1.0" encoding="UTF-8"?>

<!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.studentdetail" table="SCHOOL">

<id name="sid" column="STUD\_ID" type="integer">

<generator class="assigned"/>

</id>

<property name="sname" column="stud\_name" type="string"/>

<property name="smarks" column="stud\_marks" type="double"/>

<property name="sjoindate" column="stu\_join" type="date"/>

</class>

</hibernate-mapping> This is the heart of the entire Hibernate where programmer writes instructions to Hibernate how it should behave with **Java variables**and **database columns**. There must be a primary key in the table and the primary key here is **STUD\_ID** corresponding to **sid** in the Student bean class. Let us go in detail.

|  |  |
| --- | --- |
| ELEMENT NAME | FUNCTIONALITY |
| <HIBERNATE-MAPPING> | It is the root element of the above XML mapping file. <class> element is the child element of it. |
| <CLASS> | This element contains all the mapping information of which Java bean variable (or field or property) is to be mapped to which database column. The "name" attribute should be given the name of the Java bean class (here, it is **studentdetail**) and "table" attribute should be given the name of the database table (here, it is **school**). |
| <ID> | This element gives Hibernate the primary key information of the table. "name" attribute gives the name of the Java bean variable. The "column" attribute gives the table column name. Here the value of **sid** is to be inserted into **stud\_id** column of table **school**. **stud\_id** works as primary key. "type" attribute is given value as **integer**. The **integer** value is the data type of Hibernate and not of Java. The **integer** value converts **sid int** value to database specific SQL column type (in Oracle, it is number). |
| <GENERATOR> | This describes the primary key assignment. The primary key value can be assigned by the programmer himself or can be asked the Hibernate to assign a value by itself as per its algorithm. Here, the value assigned tells Hibernate that primary key value is assigned by the programmer himself. How to tell the Hibernate to generate itself (where programmer will not give), we will see later. |
| <PROPERTY> | This element tells Hibernate which bean field to be mapped to which table column. The "name" attribute should be given the name of the bean field (or instance variable) name and "column" attribute should be given the name of the column in datable table. In the first property, the value of **sname** of **studentdetail** class should be inserted into the**stud\_name** column of **school** table. The "type" attribute value refers Hibernate string (not Java String) which at runtime converts **sname**String value to database specific column type (in Oracle, it is varchar). |  |

The type **integer**, **string**, **double** and **date** are the data types of Hibernate not of Java which at runtime converted to database specific (here, it is Oracle) column types.

**4. Client program – StudentClient.java**

Finally we write a client program which when executed inserts records in the database table.

import java.util.Date;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class StudentClient

{

  public static void main(String[] args) throws Exception

  {                                           // create Configuration class

                                              // Configuration object parses and reads .cfg.xml file

    Configuration c = new Configuration();

    c.configure("/hibernate.cfg.xml");

                                             // SessionFactory holds cfg file properties like

                                             // driver props and hibernate props and mapping file

    SessionFactory sf=c.buildSessionFactory();

                                            // create one session means Connection

    Session s = sf.openSession();

                                            // before starting save(),update(), delete() operation we need to start TX

                                            // starting tx mean    con.setAutoCommit(false);

    Transaction tx = s.beginTransaction();

    try

    {

      studentdetail std1=new studentdetail ();

      std1.setSid(100);

      std1.setSname("S N Rao");

      std1.setSmarks(78);

      std1.setSjoindate(new Date());

      studentdetail std2=new studentdetail ();

      std2.setSid(101);

      std2.setSname("Sumathi");

      std2.setSmarks(52);

      std2.setSjoindate(new Date());

      s.save(std1);                              // stmt.addBatch("INSERT INTO school VALUES (....)");

      s.save(std2);

      s.flush(); // stmt.executeBatch()

      tx.commit(); // con.commit();

      System.out.println("Records inserted");

    }

    catch(Exception e)

    {

      tx.rollback();                            // con.rollback();

    }

  }

}

*Configuration c = new Configuration();  
c.configure(“/hibernate.cfg.xml”);*

The starting point of the client program is creating an object of **Configuration** class.**Configuration** object, here it is **c**, job is to load the **hibernate.cfg.xml**file, read the database particulars and return these particulars to **SessionFactory** object **sf**.

*Session s = sf.openSession();*

The **Session** object is equivalent to **Connection** object of JDBC. Any data to be persisted in the database is passed to **Session** object (say, save()), here it is **s**.

Creation of **Transaction** object **tx**is required for all save, delete and update operations. Select statements (reading records) do not require. **Transaction**object dictates the boundaries of transaction with **beginTransaction()** and **commit()** (or **rollback()**) methods.

*Student std1=new Student();  
std1.setSid(100);  
std1.setSname(“S N Rao”);  
std1.setSmarks(78);  
std1.setSjoindate(new Date());*

Two **Student**objects **std1** and **std2** are created and properties are set with **setXXX()**methods as declared in Student bean program.

*s.save(std1);  
s.save(std2);*

The **save()**, **update()** and **delete()**methods of **Session** class do not insert data immediately. Instead, these methods write **addBatch()** statements and keep ready for execution. The **flush()**method internally writes one **executeBatch()** statement and executes all the **addBatch()**statements. This style increases considerable performance.

*tx.commit();*

The **commit()** method of **Transaction** does commit operation on the statement executed earlier with **flush()** method.

**class attributes (in hibernate.cfg.xml file)**

Following are the most frequently used values.

1. **increment:** It increments itself and generates ID of type short, int or long.
2. **identity:**It is specific for DB2, MySQL server, Sybase etc. The generated type may be short, int or long.
3. **sequence:**To generate an ID, the database uses a sequence execution. The generated type may be short, int or long.
4. **hilo:** High-low generator internally uses a hi/lo algorithm to generate identifiers of data type short, int and long.
5. **native:** It takes some algorithm like sequence, identity or hilo specific to the database.
6. **assigned:**Here, programmer should assign the ID for himself.
7. foreign: It uses an ID of another associated object; generally used in one-to-one association.

Dialects (databases) supported by Hibernate

Hibernate supports following databases.

SQL Dialects in Hibernate

For connecting any hibernate application with the database, you must specify the SQL dialects. There are many Dialects classes defined for RDBMS in the org.hibernate.dialect package. They are as follows:

|  |  |
| --- | --- |
| **RDBMS** | **Dialect** |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle9i | org.hibernate.dialect.Oracle9iDialect |
| Oracle10g | org.hibernate.dialect.Oracle10gDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| MySQL with InnoDB | org.hibernate.dialect.MySQLInnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| DB2 | org.hibernate.dialect.DB2Dialect |
| DB2 AS/400 | org.hibernate.dialect.DB2400Dialect |
| DB2 OS390 | org.hibernate.dialect.DB2390Dialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Ingres | org.hibernate.dialect.IngresDialect |
| Progress | org.hibernate.dialect.ProgressDialect |
| Mckoi SQL | org.hibernate.dialect.MckoiDialect |
| Interbase | org.hibernate.dialect.InterbaseDialect |
| Pointbase | org.hibernate.dialect.PointbaseDialect |
| FrontBase | org.hibernate.dialect.FrontbaseDialect |
| Firebird | org.hibernate.dialect.FirebirdDialect |

## Environment Setup for Hibernate Annotation

First of all you would have to make sure that you are using JDK 5.0 otherwise you need to upgrade your JDK to JDK 5.0 to take advantage of the native support for annotations.

Second, you will need to install the Hibernate 3.x annotations distribution package, available from the sourceforge: ([Download Hibernate Annotation](http://sourceforge.net/projects/hibernate/files/hibernate-annotations/)) and copy **hibernate-annotations.jar, lib/hibernate-comons-annotations.jar** and **lib/ejb3-persistence.jar** from the Hibernate Annotations distribution to your CLASSPATH

## Annotated Class Example:

As I mentioned above while working with Hibernate Annotation all the metadata is clubbed into the POJO java file along with the code this helps the user to understand the table structure and POJO simultaneously during the development.

Consider we are going to use following EMPLOYEE table to store our objects:

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table:

import javax.persistence.\*;

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id @GeneratedValue

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private int salary;

public Employee() {}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

}

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy. Following section will explain the annotations used in the above class.

## @Entity Annotation:

The EJB 3 standard annotations are contained in the **javax.persistence**package, so we import this package as the first step. Second we used the**@Entity** annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

## @Table Annotation:

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

## @Id and @GeneratedValue Annotations:

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation which takes two parameters **strategy** and **generator** which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

## @Column Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

* **name** attribute permits the name of the column to be explicitly specified.
* **length** attribute permits the size of the column used to map a value particularly for a String value.
* **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
* **unique** attribute permits the column to be marked as containing only unique values.