Hibernate Application Requirements

Agenda

Hibernate Application Requirements :

To develop a simple hibernate application we required 4 files.

1. POJO class
2. Mapping XML
3. Configuration XML
4. Client application to write our logic

These are minimum requirement to run any Hibernate application,
and in fact we may require any number of POJO classes and any number of mapping xml files (Number of POJO classes = that many number of mapping XMLs),
and only one configuration xml and finally one java file to write our logic.

POJO class :

POJO is an acronym for Plain Old Java Object. The name is used to emphasize that a given object is an ordinary Java Object, not a special object.

The term "POJO" is mainly used to denote a java object which does not follow any of the major java object models, conventions, or frameworks.

A POJO is a java object not bound by any restriction other than those forced by the java language specification. i.e., a POJO **should not** have to

- Extend pre-specified classes, as in

```
Ex :

public class Google extends
       javax.servlet.http.HttpServlet { .. }
```

- Implement pre-specified interfaces, as in

```
Ex :

public class Bing extends
       javax.ejb.EntityBean { .. }
```

- Contain pre-specified annotations , as in

```
Ex :

@javax.persistence.Entity public class Yahoo{ .. }
```

However, due to technical difficulties and other reasons , many software products or frameworks described as POJO - compliant actually still required the use of pre-specified annotations for features such as persistence to work property.

JavaBeans :

- JavaBeans are reusable software components for java.
- Practically, they are classes written in the java programming language conforming to a particular convention.
- They are used to encapsulate many objects into a single object(the bean), so that they can be passed around as a single bean object instead of as multiple individual objects.
- A JavaBean is a java object that is serializable, has a zero-argument constructor, and allows access to properties using getter and setter method.

Mapping and Configuration files in Hibernate :

Mapping :

- Every ORM tool needs this mapping. Mapping is the mechanism of placing object properties into columns of a table.
- Mapping can be given to an ORM tool either in the form of an XML or in the form of the annotations.
- The mapping file contains mapping from a pojo class name to a table name and pojo class variable names to table column names.
- While writing a hibernate application, we can construct one or more mapping files, mean hibernate application can contain any number of mapping files.

Generally an object contains 3 properties like

1. Identity (Object Name)

2. State (Object values)
3. Behavior (Object Methods)

But while storing an object into the database, we need to store only the values (State), But how to avoid identity, behavior ? In order to inform what values of an object have to be stored in what column of the table, will be taking care by the mapping.

Mapping can be done using 2 ways :

1. XML
2. Annotations

Note : annotations are introduced into java from JDK 1.5

Syntax of mapping xml :

```
<hibernate-mapping>
<class-name name="POJO-class-name" table="table-name-in-database">
 <id name="variable-name" column="column-name-in-database"/>
 <property name="variable-name" column="column-name-in-database"/>
 <property name="variable-name" column="column-name-in-database"/>
</class-name>
</hibernate-mapping>
```

Configuration :

Configuration is the file loaded into an hibernate application when working with hibernate, this configuration file contains 3 types of information.

- Connection Properties
- Hibernate Properties
- Mapping file names

Note : We must create one configuration file for each database we are going to use, suppose if we want to connect with 2 databases, like Oracle, MySql then we must create 2 configuration files.

Syntax of Configuration xml :

```
<hibernate-configuration>

<session-factory>

<!-- Related to the connection properties -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
<property name="connection.username">lms</property>
```

```
<property name="connection.password">scott</property>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="myeclipse.connection.profile">myJdbcDriver</property>

<!-- Related to the hibernate properties -->
<property name="show_sql">true/false</property>
<property name="dialect">Database dialect</property>
<property name="hbm2ddl.auto">create/update or whatever</property>

<!-- Related to mapping -->
<mapping resource="hbm file1.xml" />
<mapping resource="hbm file2.xml" />

</session-factory>

</hibernate-configuration>
```

Example on CURD operations :

Pojo class :

```
package com.beans;

public class Employee {
private long employeeId;
private String employeeName;

public long getEmployeeId() {
        return employeeId;
}
public void setEmployeeId(long employeeId){
        this.employeeId = employeeId;
}
public String getEmployeeName(){
        return employeeName;
}
public void setEmployeeName(String employeeName){
        this.employeeName = employeeName;
}

}
```

Mapping xml for POJO :

Employee.hbm.xml

```
<?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">
```

```
<!-- Related to hibernate mapping for POJO -->

<hibernate-mapping>

<class name="com.beans.Employee" table="EMPLOYEEDB" >
  <id name="employeeId" column="EID" />
  <property name="employeeName" column="ENAME" length="15" />
</class>

</hibernate-mapping>
```

Note : If the database column name is same with java bean property name then we can ignore the column.

Configuration XML :

hibernate.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>

<!-- Related to the connection properties -->

<property name="myeclipse.connection.profile">myJdbcDriver</property>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
<property name="connection.username">lms</property>
<property name="connection.password">scott</property>


<!-- Related to the hibernate properties -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create</property>

<!-- Related to hibernate mapping -->
<mapping resource="com/hiber/Employee.hbm.xml" />

</session-factory>

</hibernate-configuration>
```

Client Application 1 :

```
package com.client;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.beans.Employee;

public class ClientAppOne {

public static void main(String ar[]){

Configuration cfg=new Configuration();
cfg.configure();

SessionFactory sf=cfg.buildSessionFactory();
Session session=sf.openSession();
Transaction tx=session.beginTransaction();

Employee e=new Employee();
e.setEmployeeId(11);
e.setEmployeeName("Charan");
session.save(e);

tx.commit();

System.out.println("Success");

}
}
```

Client Application 2 :

Note :
Before running Client Application 2 , change the value of hbm2dll.auto to 'update' in hibernate.cfg.xml file .

Otherwise it will drop the table and recreate the new tables so you may get NullPointerException.

```
package com.client;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import com.beans.Employee;

public class ClientAppTwo {

public static void main(String ar[]){

Configuration cfg=new Configuration();
cfg.configure();
```

```
SessionFactory sf=cfg.buildSessionFactory();
Session session=sf.openSession();


Employee e=(Employee)session.get(Employee.class, 1l);
System.out.println(e.getEmployeeId());
System.out.println(e.getEmployeeName());

System.out.println("---------");

}
}
```

Client Application 3 :

```
package com.client;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.beans.Employee;

public class ClientAppThree {

public static void main(String ar[]){

Configuration cfg=new Configuration();
cfg.configure();

SessionFactory sf=cfg.buildSessionFactory();
Session session=sf.openSession();
Transaction tx=session.beginTransaction();

Employee e=(Employee)session.get(Employee.class, 1l);
e.setEmployeeName("Akshay");
session.update(e);

tx.commit();

System.out.println("update");

}
}
```

Client Application 4 :

```
package com.client;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
```

```
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.beans.Employee;

public class ClientAppDelete {

public static void main(String ar[]){

Configuration cfg=new Configuration();
cfg.configure();

SessionFactory sf=cfg.buildSessionFactory();
Session session=sf.openSession();
Transaction tx=session.beginTransaction();

Employee e=(Employee)session.get(Employee.class, 1l);
session.delete(e);

tx.commit();

System.out.println("delete");

}
}
```