Hibernate For Beginners

Author(s)	**Rahul Jain** ***http://rahuldausa.wordpress.com*** http://rahuldausa.blogspot.com

Contents

1. Introduction

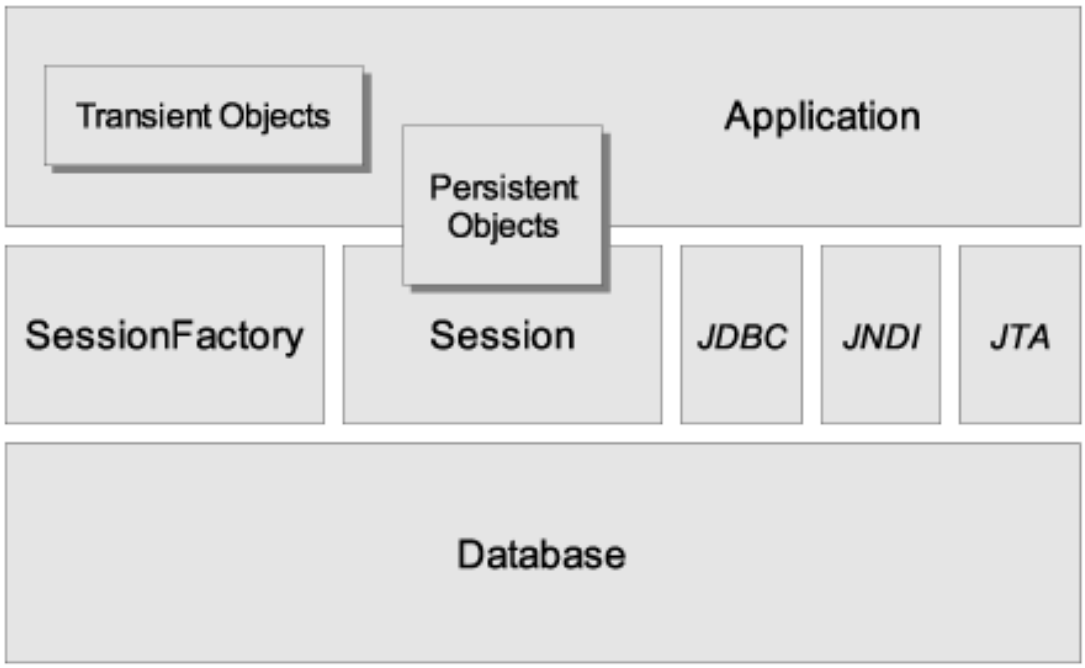
Hibernate is an open source java based library used to work with relational databases. Hibernate is an Object/Relational Mapping (ORM) tool for Java environments. The term Object/Relational Mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema. It is a very powerful ORM solution built on top of JDBC (Java Database Connectivity) API.

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. It can also significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.

Hibernate can be also configured to use a database connection pooling mechanism to improve the performance for database operation. In this connections are checked out from database and maintained in a pool and after every database operation is returned back to pool. These connections can be returned back to database after a certain period if idleness so other applications, if any would not go out of database connections. By default, hibernate uses an inbuilt connection poll, but for production based environment it is not suggested. Hibernate can be configured for Apache DBCP or C3P0, a more famous and reliable connection pooling APIs.

Hibernate makes use of persistent objects called POJOs (Plain Old Java Objects) along with XML mapping documents for persisting objects into database. POJOs are a simple Java Object has getter and setter methods for an attribute. These objects works as a data carrier called as Data Transfer Object (DTO).They are used to carry data between different layers. The data is binded to method of these classes.

2. Hibernate Architecture



Hibernate High level architecture (2.1)

Below are some definitions of the objects depicted in the 2.1 diagram:

1. **SessionFactory (org.hibernate.SessionFactory)**
    a. Session factory is a threadsafe, immutable cache of compiled mappings for a single database. A factory for Session and a client of ConnectionProvider, SessionFactory can hold an optional (second-level) cache of data that is reusable between transactions at a process, or cluster, level. Only single instance of session factory is required for an application, so it is based on a singleton pattern. SessionFactory object is loaded at the start of the application.

2. **Session (org.hibernate.Session)**
    a. Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store (database, xml). It wraps a JDBC connection and is a factory for Transaction. Session holds a mandatory first-level cache of persistent objects that are used when navigating the object graph or looking up objects by identifier.

3. **Persistent Objects and Collections**
    a. These are short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one Session. Once the Session is closed, they will be detached and free to be used in any application layer (for example, directly as data transfer objects to and from presentation).

4. **Transient and Detached Objects and Collections**
    a. Instances of persistent classes those are not currently associated with a Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed Session.

5. **Transaction (org.hibernate.Transaction)**
    a. (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A Session might span several Transactions in some cases. However, transaction demarcation, either using the underlying API or Transaction, is never optional.

6. **ConnectionProvider (org.hibernate.connection.ConnectionProvider)**
    a. (Optional) It is a factory for, and a pool of, JDBC connections. It abstracts the application from underlying Datasource or DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.

7. **TransactionFactory (org.hibernate.TransactionFactory)**
    a. (Optional) It is a factory for Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

8. ***Extension Interfaces***

    Hibernate offers a range of optional extension interfaces you can implement to customize the behavior of your persistence layer. These can be check in the API documentation for further details.

Instance States :

An instance of persistent class can be in three different states; these states are defined in relation to a *persistence context*. The Hibernate Session object is the persistence context. The three different states are as follows:

1. **Transient**
    a. The instance is not associated with any persistence context. It has no persistent identity or primary key value.

2. **Persistent**
    a. The instance is currently associated with a persistence context. It has a persistent identity (primary key value) and can have a corresponding row in the database. For a particular persistence context, Hibernate *guarantees* that persistent identity is equivalent to Java identity in relation to the in-memory location of the object.

3. **Detached**

a. The instance was once associated with persistence context, but that context was closed, or the instance was serialized to another process. It has a persistent identity and can have a corresponding row in the database. For detached instances, Hibernate does not guarantee the relationship between persistent identity and Java identity.

Transaction Management :

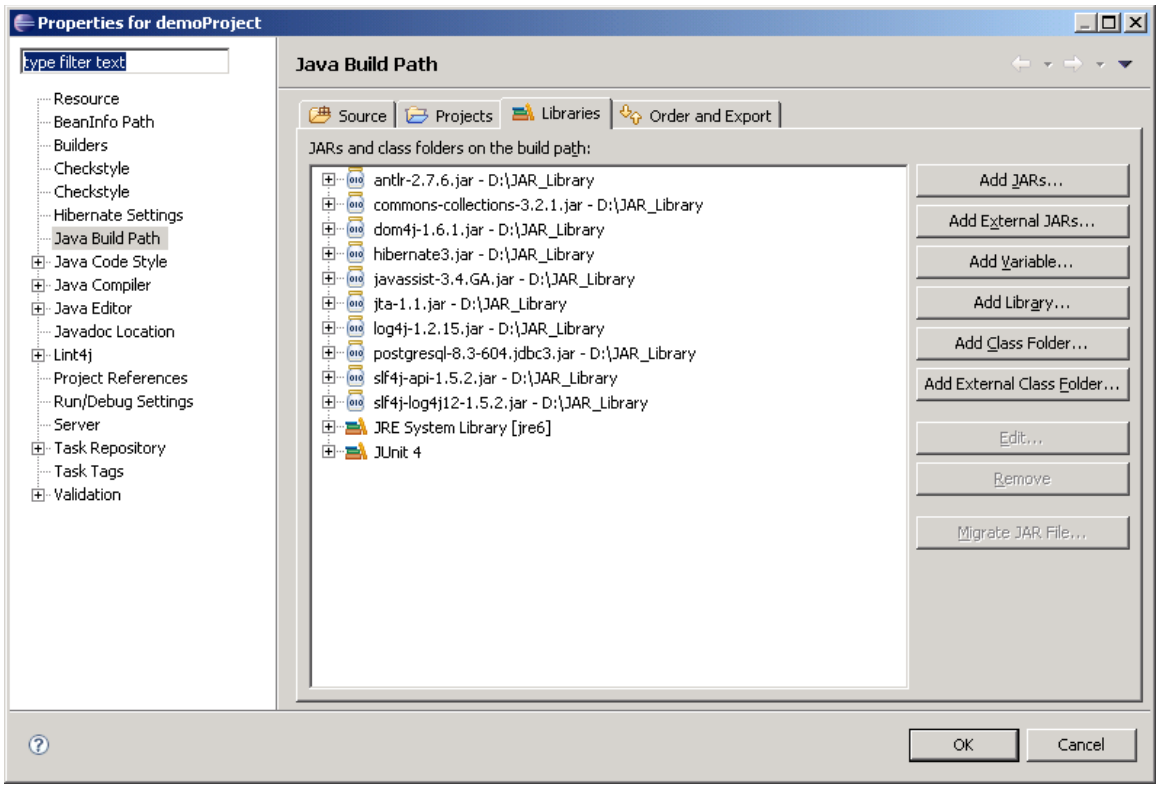
Transaction Management service provides the ability to the user to execute more than one database statement at a time. To start a new transaction **session.beginTransaction()** method should be invoked on a session object.

If transaction is successfully executed it should be committed to persist the data in database by invoking **transaction.commit() method.**

In case of any exception or failure, the transaction must be rolled back using **transaction.rollback()** to synchronize the database state, otherwise database will throw an exception as current transaction is aborted.

3. Building a Simple Application Using Hibernate

To build a simple application using hibernate, first we need to create a java project in eclipse or IDE in which you feel comfortable. After creation of a java project, add these below libraries to the project's build path. You can download the latest version of hibernate from www.hibernate.org, if you have not already got one. The version used in this tutorial is 3.0. You may find it easier to download the package containing all the dependencies so that you know that all the necessary jar files are present. Some of these libraries are used by hibernate at run time.



Now we need to create a User Details table in database to store user's detail and a database dialect in hibernate configuration file to tell hibernate to use database queries as per the database.

In this tutorial we are using PostgreSQL8.3 so org.hibernate.dialect.PostgreSQLDialect is used as database dialect in hibernate configuration file. If you are using database other than PostgreSQL you can refer the below table to find the appropriate dialects. In hibernate 3.3 only these below databases are supported by hibernate.

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect

RDBMS	Dialect
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

Courtesy : http://docs.jboss.org/hibernate/core/3.3/reference/en/html/session-configuration.html#configuration-optional-dialects

Below is the SQL script for creating a User Details table in database. This below SQL script is according to PostGresSQL8.3.1, so if you are using a database other than PostgreSQL you need to write a similar one as per your database.

User_Details.sql : We will use the below SQL query to create the table in the database.

```
CREATE TABLE user_details (
  user_id integer NOT NULL,
  user_name character varying(20),
  user_password character varying(20),
  CONSTRAINT "USER_pkey" PRIMARY KEY (user_id)
)
```

4. Application Configuration – hibernate.cfg.xml

The hibernate.cfg.xml is a configuration file that is used for defining the database parameters such as database username, password, connection URL, database driver class, connection provider, and connection pool size. This also includes hibernate mapping files (hbm), a xml based hibernate mapping file that have mapping of database column to java attributes.

Below is a sample hibernate configuration file.

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>
                <property name="hibernate.connection.url">jdbc:postgresql://127.0.0.1:5432/qc</property>
                <property name="hibernate.connection.username">postgres</property>
                <property name="hibernate.connection.password">postgres</property>
                <property name="hibernate.connection.pool_size">10</property>
                <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
                <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
                <property name="current_session_context_class">thread</property>
                <property
name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
                <property name="show_sql">true</property>
                <mapping resource="com/hibernatetest/demo/User.hbm.xml" />
        </session-factory>
</hibernate-configuration>
```

These configurations are used by hibernate to build session factory i.e. creation of database connections.

a. **hibernate.connection.url** : This is the database Connection URL signifies the database URL e.g. jdbc:postgresql://127.0.0.1:5432/qc
b. **hibernate.connection.username** : database username e.g. postgres
c. **hibernate.connection.password** : database password e.g. postgres
d. **hibernate.connection.pool_size** : database connection pool size. It signifies that this number of connections will be checked out by application at the time of creation of session factory.
e. **hibernate.connection.driver_class** : Database Driver class e.g. for PostgreSQL it is org.postgresql.Driver
f. **hibernate.dialect** : Dialect to tell hibernate to do syntax conversion for database queries based on the database. e.g. org.hibernate.dialect.PostgreSQLDialect
g. **current_session_context_class** : context in which current session need to be maintained. e.g. thread
h. **transaction.factory_class** : transaction factory e.g. org.hibernate.transaction.JDBCTransactionFactory>
i. **show_sql** : Should Hibernate show SQL for every database operation. Very good for debugging purpose. In production environment it should be tuned off. e.g. true/false
j. mapping resource="com/hibernatetest/demo/User.hbm.xml" : Mapping of hbm files.

User.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">
<hibernate-mapping>
      <class name="com.hibernatetest.demo.User" table="user_details"
schema="public" optimistic-lock="version">
            <id name="userId" type="java.lang.Integer">
                  <column name="user_id" />
                  <generator class="increment" />
            </id>
            <property name="userName" type="string">
                  <column name="user_name" length="20" />
            </property>
            <property name="userPassword" type="string">
                  <column name="user_password" length="20" />
            </property>
      </class>

</hibernate-mapping>
```

Below is the description for above xml file.

1. **Class** : name of the class (pojo) with package.
   Table : name of the database table
   Schema : name of `the database schema.`
2. **Optimistic-lock** : It is a kind of locking mechanism but in real it does not lock any row or column in database. It signifies a kind of database row state checker, that tell if the row that is getting modified by current transaction is already updated by another transaction after its read from database. Lets we take an example to get it understand in more detail.

   1. Let say a user want to book an air ticket. On booking portal he found that only one ticket is available. He starts booking the ticket. In this duration an another user also came on the same web portal and he also start booking the ticket for same flight, in this case only one user would be able to book the ticket for booking as for one it would fail. Actually this issue started as when the first user is booking the ticket, but still system is allowing to book ticket for another user when only one ticket is left. This is called the dirty read.

   2. To overcome this kind of situation, hibernate use a versioning check so it can know that this record is already updated when other users is also trying to updating the same record at the same time.

   3. For this a field named as version is maintained in table. When user updates a record, it checks the version value in database with the getting updated record is same or not. If it is same, then it will allow updating the database and increment the version counter otherwise it will throw a stale record exception. As if in the duration of this process, some other user also read and try to update the same record, the version counter would be different than the database one.

Generator : This is used to update the primary key. This tells hibernate to use which strategy to update the primary key.

Mainly these below strategies are used to update the primary key.

1. assigned : It signifies a user itself provide the value for primary key. This is not suggested to use in a cluster based application.
2. increment : hibernate will increment the primary key automatically.
3. sequence : A sequence defined in db can be used to auto increment the primary key.

Property : attribute of pojo class, this is mapped to database column name.

HibernateUtil.java : It is a java class that returns a session factory object that is used for getting the session(connection) object.

```
package com.hibernatetest.demo;


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


/**
 * @author rjain6
 */
public class HibernateUtil
{

   private static SessionFactory sessionfactory = null;

    public static SessionFactory getSessionFactory()
    {
        Configuration cfg = new Configuration();
        sessionfactory =
cfg.configure("com/hibernatetest/demo/hibernate.cfg.xml").buildSessionFactory();
        return sessionfactory;
    }

}
```

5. Creating the POJO(Plain Old Java Objects)

Now for persisting data to the database, hibernate requires a Java bean (POJO) object to be passed. For this we are writing a User's pojo object having all getter and setter methods of attributes. These attributes will be work as carrier of data that will be stored in database. As these POJO Objects will travel across the network, these objects should implement the `Serializable` interface.

```
package com.hibernatetest.demo;


import java.io.Serializable;
```

```
/**
 * This class is a POJO that works as a carrier of the
 * data and will be stored in database by hibernate.
 * @author rjain6
 */
public class User implements Serializable
{

    private int userId;
    private String userName;
    private String userPassword;

    public int getUserId()
    {
        return userId;
    }

    public void setUserId(int userId)
    {
        this.userId = userId;
    }

    public String getUserName()
    {
        return userName;
    }

    public String getUserPassword()
    {
        return userPassword;
    }

    public void setUserName(String userName)
    {
        this.userName = userName;
    }

    public void setUserPassword(String userPassword)
    {
        this.userPassword = userPassword;
    }

}
```

6. Creating the DAO (Data Access Objects)

Since this is a very simple application, here only one DAO (Data Access object) is involved. This Dao class has methods to perform all the basic CRUD (Create, Read, Update, and Delete) functionalities.

```
package com.hibernatetest.demo;


import java.io.Serializable;
import java.util.List;
```

```
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;


/**
* This is DAO class for performing database related functionalities.
 * @author rjain6
 */
public class UserDAO
{

/**
 * Creating a user in the database
 */
    public int createRecord(User user)
    {
        Transaction tx = null;
        Session session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        Serializable s = session.save(user);
        tx.commit();
        return ((Integer)s).intValue();
    }

/**
 * Updating a user's details in the database
 */
    public boolean updateRecord(User user)
    {
        boolean isSuccess = false;
        Transaction tx = null;
        Session session = HibernateUtil.getSessionFactory().openSession();
        try
        {
            tx = session.beginTransaction();
            session.update(user);
            tx.commit();
            isSuccess = true;
        } catch (HibernateException e)
        {
            isSuccess = false;
        }
        return isSuccess;
    }

/**
 * Deleting the user from the database
 */
    public boolean deleteRecord(User user)
    {
        boolean isSuccess = false;
        Transaction tx = null;
        Session session = HibernateUtil.getSessionFactory().openSession();
        try
```

```
        {
            tx = session.beginTransaction();
            session.delete(user);
            tx.commit();
            isSuccess = true;
        } catch (HibernateException e)
        {
            isSuccess = false;
        }
        return isSuccess;
    }

/**
 * Retrieving All users' details from the database
 */
    public List retrieveAllRecord()
    {
        Transaction tx = null;
        Session session = HibernateUtil.getSessionFactory().openSession();
        List ls = null;
        tx = session.beginTransaction();
        Query q = session.createQuery("from User");
        ls = q.list();
        return ls;
    }

/**
 * Retrieving All users' names from the database
 */
    public List retrieveAllUserName()
    {
        Transaction tx = null;
        Session session = HibernateUtil.getSessionFactory().openSession();
        List ls = null;
        tx = session.beginTransaction();
        //Native Query
        Query q = session.createSQLQuery("select user_name from User_Details");
        ls = q.list();
        return ls;
    }

}
```

7. Testing the DAO functionality using JUnit Test

The final stage in this simple tutorial is to create Junit test class to check the Session factory object and functionality of the data layer.

The contents of the class are shown below.

`HibernateUtilTest`.java : It is Test class for testing the working of session factory object. If Session factory object is not null, that means hibernate is able to get the database connection from DB.

```
package com.hibernatetest.demo;

import org.junit.Test;


/**
 * This is a test class for chcecking is sessionfactory(database connection) is
loaded by hibernate properly.
 * @author rjain6
 */
public class HibernateUtilTest
{

    /**
     * Test method for {@link
com.hibernatetest.demo.HibernateUtil#getSessionFactory()}.
     */
    @Test
    public void testGetSessionFactory()
    {
        System.out.println("session factory:"
                + HibernateUtil.getSessionFactory());
    }

}
```

UserDAOTest.java :

```
package com.hibernatetest.demo;


import java.util.Collection;
import java.util.List;

import org.junit.Test;


/**

* This is test class having test methods to check the functionality of  UserDAO
class .

 * @author rjain6
 */
public class UserDAOTest
{

    @Test
    public void createRecordTest()
    {
        UserDAO userDAO = new UserDAO();
        User user = new User();
        user.setUserName("demo");
        user.setUserPassword("demo");
        int recordId = userDAO.createRecord(user);
```

```
        System.out.println("recordId:" + recordId);
    }

    @Test
    public void updateRecordTest()
    {
        UserDAO userDAO = new UserDAO();
        User user = new User();
        user.setUserId(2);
        user.setUserName("demo123");
        user.setUserPassword("demo123");
        boolean status = userDAO.updateRecord(user);
        System.out.println("status:" + status);
    }

    @Test
    public void deleteRecordTest()
    {
        UserDAO userDAO = new UserDAO();
        User user = new User();
        user.setUserId(3);
        boolean status = userDAO.deleteRecord(user);
        System.out.println("status:" + status);
    }

    @Test
    public void retrieveRecordTest()
    {
        UserDAO userDAO = new UserDAO();
        List list = userDAO.retrieveAllRecord();
        if (isNullSafe(list))
        {
            for (int i = 0;i < list.size();i++)
            {
                System.out.println("UserName:" +((User)list.get(i)).getUserName());
                System.out.println("UserPassord:" +
((User)list.get(i)).getUserPassword());
            }
        }
    }

    @Test
    public void retrieveAllUserNameTest()
    {
        UserDAO userDAO = new UserDAO();
        List ls = userDAO.retrieveAllUserName();
        if (isNullSafe(ls))
        {
            for (int i = 0;i < ls.size();i++)
            {
                System.out.println("UserName:" + ls.get(i));
            }
        }
    }

    /**
```

```
     * @param ls
     * @return
     */
    private boolean isNullSafe(Collection col)
    {
        if (col != null && col.size() > 0)
            return true;
        else
            return false;
    }
}.
```

8. Conclusion

Hibernate is an ORM tool that is used to map the database structures to java objects at run time. Using a persistence framework like Hibernate allows developers to focus on writing business logic code instead of writing an accurate and good persistence layer which include writing the SQL Queries, JDBC Code , connection management etc.

9. References

1. https://www.hibernate.org/
2. http://docs.jboss.org/hibernate/core/3.3/reference/en/html/architecture.html

10. Glosssary

- ORM : Obejct Relational Mapping
- SQL : Structural Query Language
- HQL : Hibernate Query Language
- POJO : Plain Old Java Objects
- JDBC : Java Database Connectivity API
- HBM : Hibernate Mapping