**Hibernate**

Draw Backs of JDBC:

* In JDBC, if we open a database connection we need to write in try, and if any exceptions occurred catch block will takers about it, and finally used to close the connections.
* here as a programmer we must close the connection, or we may get a chance to get our of connections message…!
* Actually if we didn’t close the connection in the finally block, then jdbc doesn’t responsible to close that connection.
* In JDBC we need to write Sql commands in various places, after the program has created if the table structure is modified then the JDBC program doesn’t work, again we need to modify and compile and re-deploy required, which is tedious.
* JDBC used to generate database related error codes if an exception will occurs, but java programmers are unknown about this error codes right.
* In the Enterprise applications, the data flow with in an application from class to class will be in the form of objects, but while storing data finally in a database using JDBC then that object will be converted into text.  Because JDBC doesn’t transfer objects directly.

In order to overcome above problems,  Hibernate came into picture..!

# Prerequisites:

# Good understanding of the Java programming language. A basic understanding of relational databases, JDBC and SQL

# ****What is Hibernate?****

# ****Hibernate**** is a powerful and high-performance ****ORM**** service.

Hibernate maps Java classes to database tables and from Java data types to SQL data types.

Hibernate framework simplifies the development of java application to interact with the database.

Hibernate maps the Java classes to the database tables.

It also provides the data query and retrieval facilities that significantly reduces the development time.

# ****ORM**** 🡪 "Object-relational Mapping".

# It is simply ****storing a Java object into a (relational) database****. When stored, Hibernate automatically maps (places) and stores each instance variable value of a Java object in a database column.

# For example, emp object of ****Employee**** class has three properties (instance variables) like ****emp.empid****, ****emp.name**** and ****emp.department****. When the object ****emp**** is stored in the database ****table**** EMPLOYEE, Hibernate stores these variables values in the columns ****EMPID****, ****NAME**** and ****DEPARTMENT**** of a database table EMPLOYEE. This reduces the code of JDBC to a maximum extent.

# ORM Overview

Object Oriented Programming use Classes whereas Relational Database use tables. In programming this will creates a gap. This gap is called impedance mismatch.

We can bridge the gap between Object oriented model and Relational model is known as Object Relational Mapping(ORM)

ORM is a mapping between Object oriented model and Relational model.

# ****The Object-Relational Impedance Mismatch:****

Object-Relational Impedance Mismatch' (sometimes called the 'paradigm mismatch') is just a fancy way of saying that object models and relational models do not work very well together. RDBMSs represent data in a tabular format , whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects.

# ****Solution of Impedance Mismatch:****

Use an ORM mapping tool which will provide a simple API for storing and retrieving Java objects directly from database.  
There are several good ORM tools available in market that will do mapping between Objects and Database tables and solve the impedance mismatch.

Following are ORM tools:

1. Hibernate
2. OpenJPA
3. EJB Entity Bean
4. EclipseLink
5. TopLink

# Java ****variables vs table columns**** mapping is done in an ****XML**** file (like web.xml, in case of servlets, where an alias name is mapped to the actual servlet).

# **Hibernate** is Ob**ject/Relational persistence** for **Java** and **.NET** technologies. Here, **persistence means writing Java object to a permanent storage and making it to live (stay) long**.

# Always persistence storage comes with writing to a hard disk like to a text file or a RDBMS table.

# Remember DS like an ****ArrayList**** or ****Vector**** gives a temporary storage (available to program until program terminates. Once the program execution is over, the data is lost and when you restart the same program, the data is not available) of data as the data is written to the RAM.

# **Object/Relational means storing or writing a Java object to a database (to be more precisely, relational database) like Oracle**.

# Storing the variables of an object is known as storing the ****state**** of the object or persisting the object.

# Hibernate is an open source code that can be freely downloaded from ****www.hibernate.org**** and used.

# Hibernate stable versions

# Hibernate 4.2.21.Final,

# Hibernate 4.3.11.Final

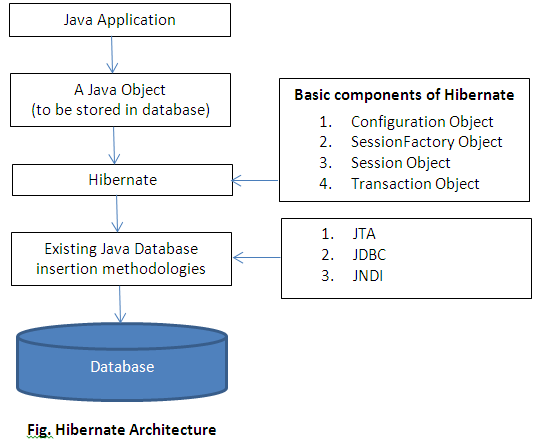
# Hibernate 5.0.12.Final

# Hibernate 5.1.4.Final

# Hibernate 5.2.7.Final

**ORM** is a supplement to **JDBC** and not a replacement. In fact, **Hibernate internally uses JDBC**. The aim of ORM is to reduce the writing of number of lines of DB programming code in Java.

**Hibernate Architecture**



Hibernate uses JDBC internally to insert a Java object into database. The main objects of Hibernate used in coding are **Configuration**, **SessionFactory**, **Session** and **Transaction**. The insertion of an object into database is known as **persisting** (literal meaning: preserving) a Java object.

**JDBC vs Hibernate – Features of Hibernate (advantages of Hibernate over JDBC)**

**1. Transparent Persistence**

It is a very complex task to write an object of an Object-Oriented programming language to a relational database with JDBC as there will be a mismatch between the **data types** (say, int) of a Java class and **column types** (say, number) of a database, and moreover, column types vary with each database. Hibernate takes care of this transparently depending on the database used (internally) without any botheration to the programmer.

Writing a Java object to a database and reading back is known as **transparent persistence**. Transparent means unknown to the programmer, to say, the internal implementation is unknown

**Another example:**

What is happening internally to store a Java object into a table is completely transparent (not known or abstracted) to the programmer. Moreover

1. How Java types like **int** and String are converted into database specific **number** and **varchar2** are not known.
2. How a Java **object is mapped to a database** table is not known to the programmer.
3. What **SQL query** is used by the Hibernate is not known (of course, this can be known with extra line of code with the option show\_aql).
4. Programmer is not aware of the internal mechanism of maintaining the **transactions**, **catches** and fetches etc.

All the above transparent mechanism amounts to less JDBC code to the programmer.

**2. Object relation management (Class – Class mapping)**

Imagine there are two Java classes **Student** and **Address** with database tables **Student** and **Addressbook**. The primary key of **addressbook** is a foreign key in **student**. When a **Student** object is inserted in **Student** table, automatically the address should also be inserted in **addressbook** table. When a **Student** record is deleted, its corresponding address in the **addressbook** table should also be deleted and so also for updations.

**3. Auto schema/Table generation**

When a Java object is tried to save to a database table, if the table does not exist, **Hibernate** creates a new table with appropriate types (specific to that database) and inserts the object. In Hibernate, inserting object means writing each variable of the Java object to each column of a table. That is, every Java object corresponds to one record in a database table.

**4. Automatic Primary key generations**

Hibernate has its own style of **primary key generation** for the records inserted without depending on the database native algorithm. Hibernate comes with its own data types (which are independent of any database). If required, programmer can suggest Hibernate to depend on the native database style or any **sequence algorithm** etc.

**5. Support for Query Language (database independent queries)**

JDBC uses **SQL** (Structured Query Language) which may vary with the database. Programmer should write an effective way of query, keeping in mind of the performance (like by decreasing the number of database hits to execute the query). Even though, Hibernate supports native **SQL**, it comes with its own query language known as **HQL (Hibernate Query Language)** which is independent of any database. **HQL** syntax is similar to SQL syntax and supports polymorphic queries. To say, **HQL** gives database independent code (SQL is database dependent code). Hibernate Query Language permits to write a query on class objects.

**6. Maintenance Cost**

JDBC programmer should write a lot of code to create Java persistent objects out of the **Resultset** object, if required. He may be required to write "emp.ename = res.getString(“empname”)" etc. where **emp** is an object of **Employee** class with variable ename. That is, the programmer should map manually himself the variables with database columns. Hibernate does this implicitly. It lessens the programmer code and this cuts short the development time and also maintenance cost.

**7. Connection pooling**

Hibernate supports connection pooling to increase the performance of database access (hits). Hibernate comes with three styles of pooling – **apache dbcp**, **proxoo**l and **C3P0**.

**8. Optimize Performance (Caching)**

If the client requires the same records again and again, Hibernate uses **cache memory**; places the records in **cache** and gives to the client. **Cache is nothing but retention (storing or accumulating) of data**. Hibernate comes with **Hibernate Dual Layer Caching Architecture (HDLCA)**. Caching is transparent to client and increases the performance. In JDBC, the programmer should hand code everything.

**9. Supports Transactions**

Hibernate comes with transaction facility. To look after, there is a **Transaction** class. Using Transaction class, a programmer can **begin** or **commit** or **rollback** a transaction.

**10. Criteria and Criterion/Conditions and Condition**

Hibernate comes with Criteria class to use with **HQL**. Criteria impose restrictions (constraints) on a database query.

**11. Open Source Code**

Hibernate is absolutely free either for trial use or commercial use. It is available under **LGPL (Lesser General Public License)**. Hibernate can be freely distributed and customized for both development and production deployments.

**12. Automatic Versioning and Time Stamping**

**Versioning** is a feature of a database where the user is guaranteed of changes done by him are not rolled back by another intentionally or unintentionally. **Automatic versioning is a technique of assuring data integrity**. Hibernate allows the programmer to specify the version field. On this field, the Hibernate updates the table each time relational tuple is modified through a Java class object. A row in a table is called **tuple** of the relation. When one user does an update operation, the other is not allowed. JDBC does not come with implicit check of updation of data.

**13. Enterprise Reliability and Scalability**

Hibernate can scale database hits from few hundreds of in-house Intranet to enterprise mission critical operations with lakhs of hits. In JDBC it is very difficult.

Disadvantages

 Hibernate increases complexity to enter data in a database with all mapping XML files etc. When database does not change frequently, I feel, JDBC is the best choice.

 In Web (Internet) environment, Hibernate API support is not adequate (acceptable/enough) .

 Hibernate does not allow to store or save two objects in a database with a single query.

# Hibernate Example with Eclipse IDE

1) Create the java project

2) Add jar files for hibernate and also add jdbc jar file to the project for jdbc Driver class “ojdbc14-1.0.jar”

3) Create the Persistent class

**package** com.visix.mnrao.emp;

**public** **class** Employee

{

**private** **int** empNum;

**private** String empName;

**private** **double** empSalary;

**private** **int** deptNum;

**public** **int** getEmpNum()

{

**return** empNum;

}

**public** **void** setEmpNum(**int** empNum)

{

**this**.empNum = empNum;

}

**public** String getEmpName()

{

**return** empName;

}

**public** **void** setEmpName(String empName)

{

**this**.empName = empName;

}

**public** **double** getEmpSalary()

{

**return** empSalary;

}

**public** **void** setEmpSalary(**double** empSalary)

{

**this**.empSalary = empSalary;

}

**public** **int** getDeptNum()

{

**return** deptNum;

}

**public** **void** setDeptNum(**int** deptNum)

{

**this**.deptNum = deptNum;

}

}

4) Create the Hibernate mapping file for Persistent class

To create the hibernate mapping file, Right click on src - new - file - specify the file name (e.g. employee.hbm.xml) - ok. It must be outside the package.

**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">

<hibernate-mapping>

<class name=*"com.visix.mnrao.emp.Employee"* table=*"EMPLOYEE"*>

<id name=*"empNum"*>

<generator class=*"assigned"*></generator>

</id>

<property name=*"empName"*></property>

<property name=*"empSalary"*></property>

<property name=*"deptNum"*></property>

</class>

</hibernate-mapping>

5) Create the Hibernate Configuration file

To create the configuration file, right click on src - new - file. Now specify the configuration file name e.g. 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=*"hbm2ddl.auto"*>update</property>

<property name=*"dialect"*>org.hibernate.dialect.Oracle9Dialect</property>

<property name=*"connection.url"*>jdbc:oracle:thin:@localhost:1521:orcl</property>

<property name=*"connection.username"*>scott</property>

<property name=*"connection.password"*>tiger</property>

<property name=*"connection.driver\_class"*>oracle.jdbc.driver.OracleDriver</property>

<mapping resource=*"employee.hbm.xml"*/>

</session-factory>

</hibernate-configuration>

### 6) Create the class that retrieves or stores the persistent object

Storing the employee object to the database.

**package** com.visix.mnrao.emp;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** StoreData

{

**public** **static** **void** main(String[] args)

{

Configuration cfg=**new** Configuration();

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

SessionFactory factory=cfg.buildSessionFactory();

Session session=factory.openSession();

Transaction t=session.beginTransaction();

Employee e1=**new** Employee();

e1.setEmpNum(1002);

e1.setEmpName("xyz");

e1.setEmpSalary(6000);

e1.setDeptNum(20);

session.persist(e1);//persisting the object

t.commit();//transaction is commited

session.close();

System.***out***.println("successfully saved");

}

}

Right click on main method 🡪 Run AS 🡪 Java Application

O/p

**Successfully saved**

**Sql>** SELECT table\_name FROM user\_tables;

Now you can see **EMPLOYEE** table in the scott schema;

**Configuration Object:**

The Configuration object is the first Hibernate object you create in any Hibernate application and usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate. The Configuration object provides two keys components:

* **Database Connection:**

This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.

* **Class Mapping Setup**

This component creates the connection between the Java classes and database tables..

**SessionFactory Object:**

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is heavyweight object so usually it is created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So if you are using multiple databases then you would have to create multiple SessionFactory objects.

**Session Object:**

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

**Transaction Object:**

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

**Hibernate Configuration File**

Hibernate Configuration File(cfg file) is the file loaded into an hibernate application when working with hibernate. Hibernate uses this file to establish connection to the database server. It is an XML file which is used to define below information. Standard name for this file is hibernate.cfg.xml

There must be one configuration file for each database used in the application, suppose if we want to connect with 2 databases, like Oracle, MySql, then we must create 2 configuration files with different names, like oracle.cfg.xml for Oracle DB and mysql.cfg.xml for mysql database.

**Sample Configuration File :**

<hibernate-configuration>

<session-factory>

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

<property name=*"dialect"*>Database dialect class</property>

<property name=*"connection.driver\_class"*>Driver Class Name </property>

<property name=*"connection.url"*>URL </property>

<property name=*"connection.user"*>USER NAME </property>

<property name=*"connection.password"*>PASSWORD</property>

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

<! -- Related to hibernate properties START -->

<property name=*"show\_sql"*>true/false</property>

<property name=*"format\_sql"*>true/false</property>

<property name=*"use\_sql\_comments"*>true/false</property>

<property name=*"hbm2ddl.auto"*>create/create-drop/update/validate</property>

<property name=*"connection.pool\_size"*>10</property>

<! -- Related to hibernate properties END-->

<! -- Related to mapping START-->

<mapping resource=*"hbm file 1 name .xml"* / >

<mapping resource=*"hbm file 2 name .xml"* / >

<! -- Related to the mapping END -->

</session-factory>

</hibernate-configuration>

**Hibernate Properties:**

**hibernate.dialect**

<property name=*"dialect"*>Database dialect class</property>

This property makes Hibernate generate the appropriate SQL for the chosen database.

**hibernate.connection.driver\_class**

<property name=*"connection.driver\_class"*>Driver Class Name </property>

The JDBC driver class.

**hibernate.connection.url**

<property name=*"connection.url"*>URL </property>

The JDBC URL to the database instance.

**hibernate.connection.username**

<property name=*"connection.user"*>USER NAME </property>

The database username.

**hibernate.connection.password**

<property name=*"connection.password"*>PASSWORD</property>

The database password.

**hibernate.connection.pool\_size**

<property name=*"connection.pool\_size"*>10</property>

Limits the number of connections waiting in the Hibernate database connection pool.

**hibernate.show\_sql**

<property name=*"show\_sql"*>true/false</property>

If the value is true, We can see generated sql statements in console

**hibernate.format\_sql :**

<property name=*"format\_sql"*>true/false</property>

If the value is true, We can see generated sql statements in a readable format

**hibernate .use\_sql\_comments :**

<property name=*"use\_sql\_comments"*>true/false</property>

If the value is true, We can see comments in generated sql statements.

**hbm2ddl.auto**

<property name=*"hbm2ddl.auto"*>create/create-drop/update/validate</property>

 **create:**  Creates schema, destroys previous data.

 **create-drop:** Drops the schema at the end of a session.

 **update:**  Updates the schema.

 **validate:** Validates the schema. It makes no changes to database

**hibernate.connection.autocommit**

Allows autocommit mode to be used for the JDBC connection.

|  |  |
| --- | --- |
| **Database** | **Dialect Property** |
| DB2 | org.hibernate.dialect.DB2Dialect |
| HSQLDB | org.hibernate.dialect.HSQLDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| Ingres | org.hibernate.dialect.IngresDialect |
| Interbase | org.hibernate.dialect.InterbaseDialect |
| Microsoft SQL Server 2000 | org.hibernate.dialect.SQLServerDialect |
| Microsoft SQL Server 2005 | org.hibernate.dialect.SQLServer2005Dialect |
| Microsoft SQL Server 2008 | org.hibernate.dialect.SQLServer2008Dialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle 11g | org.hibernate.dialect.Oracle10gDialect |
| Oracle 10g | org.hibernate.dialect.Oracle10gDialect |
| Oracle 9i | org.hibernate.dialect.Oracle9iDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| Progress | org.hibernate.dialect.ProgressDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |