**HIBERNATE**

**Hibernate Framework**

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001.

It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

**Hibernate Advantages**

1. Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
2. Provides simple APIs for storing and retrieving Java objects directly to and from the database.
3. If there is change in the database or in any table, then you need to change the XML file properties only.
4. Hibernate does not require an application server to operate.
5. Manipulates Complex associations of objects of your database.
6. Minimizes database access with smart fetching strategies.
7. Provides simple querying of data.

**Why ORM?**

* When we work with an object-oriented system, there is a mismatch between the object model and the relational database.
* RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.

**ORM (O**bject **R**elational **M**apping**)**

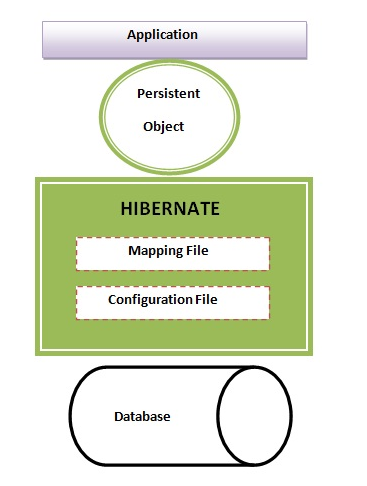
* It is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

**Advantages of ORM**

* Code access objects rather than DB tables.
* It hides details of SQL queries from OO logic.
* It is based on JDBC 'under the hood.'
* No need to deal with the database implementation.
* Transaction management and automatic key generation.
* Fast development of application.

**Hibernate Architecture**

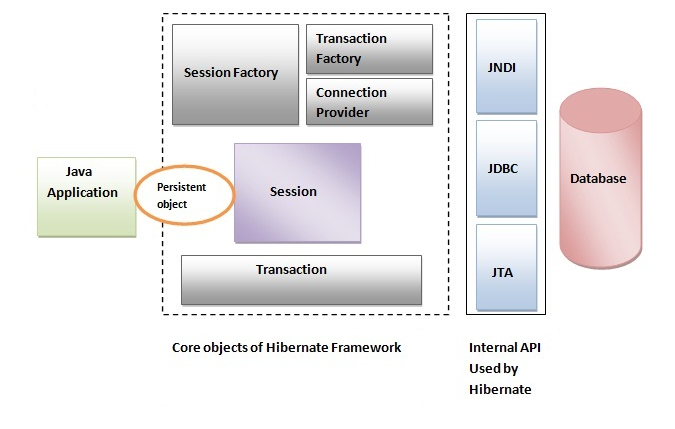
* Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns



**High level Architecture of Hibernate**

The Hibernate architecture includes many objects persistent object, session factory, transaction factory, connection factory, session, transaction etc.

There are 4 layers in hibernate architecture java application layer, hibernate framework layer, backend API layer and database layer.



#### org.hibernate.cfg.Configuration (It is class)

This object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

It is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file.

The Configuration object provides two keys components:

1. **Database Connection:**

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

1. **Class Mapping Setup:**

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

**Methods:-**

Public Configuration configure(String resource);

Public SessionFactory buildSessionFactory();

#### org.hibernate.SessionFactory (interface)

It is an interface which provides factory method to get the object of Session.

**Methods:-**

Public Session openSession();

Public Session getCurrentSession();

#### org.hibernate.Session (interface)

#### A Session is used to get a physical connection with a database. The session object provides an interface between the application and data stored in the database. It provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

**Methods:**

1. **Public Transaction beginTransaction();**

🡪 Begins unit of work.

1. **Public Transaction getTransaction();**

🡪 Get the Transaction instance associated with this session.

1. **Public Serializable save(Object);**
2. **Public void persist(Object);**
3. **Public Object get(Class,Serializable);** 🡪 eg:-get(Employee.class,101);
4. **Public void update(Object);**
5. **Public void delete(Object);**
6. **Public Query createQuery(String query);**

* Create a new instance of Query for the given HQL query string

1. **Public SQLQuery createSQLQuery(String query);**

* Create a new instance of SQL Query for the given SQL query string

1. **Public Criteria createCriteria(Class persistentClass);**

* Create a new Criteria instance, for the given entity class.

#### org.hibernate.Transaction (interface)

#### The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

**Methods:-**

Public void begin();

Public void commit();

Public void rollback();

#### org.hibernate.connection.ConnectionProvider (interface)

#### It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

#### org.hibernate.transaction.TransactionFactory (interface)

#### It is a factory of Transaction. It is optional.

**Hibernate Configuration File**

Hibernate requires a set of configuration settings related to database and other related parameters. Hibernate configuration can be done through standard java properties file called as hibernate.properties or as an XMl file named as hibernate.cfg.xml

**Important hibernate properties:-**

1. hibernate.dialect

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

e.g.:-

org.hibernate.dialect.OracleDialect

org.hibernate.dialect.MySQLDialect

org.hibernate.dialect.DB2Dialect

org.hibernate.dialect.PostgreSQLDialect

1. hibernate.connection.driver\_class 🡪 To specify the JDBC driver class.
2. hibernate.connection.url 🡪 The JDBC URL to the database.
3. hibernate.connection.username 🡪 The database username.
4. hibernate.connection.password 🡪 The database password.

E.g.:-

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

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

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

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/sasi</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password"></property>

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

</session-factory>

</hibernate-configuration>

**Hibernate Mapping file**

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate — how to map the defined class to the database table.

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

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-mapping>

<class name=”classname” table=”tablename”>

<id name=”” column=”” type=””><generator class=””/></id>

<property name=”” column=”” type=””/>

<property name=”” column=”” type=””/>

</class>

</hibernate-mapping>

**HIBERNATE – ANNOTATIONS**

Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. 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. All standard annotations are contained in the javax.persistence package.

**@Entity Annotation**

It 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**

It allows you to specify the details of the table that will be used to persist the entity in the database. It 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.

**@Id Annotation**

Each entity bean will have a primary key, which you annotate on the class with this annotation.

**@Column Annotation**

It is used to specify the details of the column to which a field or property will be mapped.

**HIBERNATE – QUERY LANGUAGE (HQL)**

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

#### org.hibernate.Query (interface)

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.

A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

**Method:-**

Public List list();

**FROM clause**

It loads complete persistent objects into memory.

E.g.

Query qry=sess.createQuery(“from Bank”);

List l1=qry.list();

**WHERE clause with Parameter**

Query qry=sess.createQuery (“from Bank where accno=:acc\_no”);

Qry.setParameter(“acc\_no”,10001);

List l1=qry.list();

Note:-

We can do this kind of process for **SELECT** clause, **ORDER** **BY** clause, **GROUP** **BY** clause, **UPDATE** clause, **INSERT** clause and **DELETE** clause.

**Aggregate Methods**

HQL supports a range of aggregate methods, similar to SQL.

E.g.

Query qry = session. createQuery(“select count(amt) from Bank”);

**Pagination using Query**

**Methods:-**

**Public Query setFirstResult(int startPosition)**

🡪 This method takes an integer that represents the first row in your result set, starting with row 0.

**Public Query setMaxResults(int maxresult)**

**-**🡪 This method tells Hibernate to retrieve a fixed number maxResults of objects.

**HIBERNATE – CRITERIA QUERIES**

Hibernate provides Criteria API, which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

Eg:-

Criteria cr=sess.createCriteria(Employee.class);

List l1=cr.list();

**Restrictions with Criteria**

Cr.add(Restrictions.gt(“salary”,25000));

**Sorting Results**

Cr.addOrder(Order.asc(“salary”));

**Projections & Aggregations**

Cr.setProjection(Projections.sum(“salary”));

Cr.setProjection(Projections.avg(“salary”));

**Pagination using Criteria**

Cr.setFirstResult(1);

Cr.setMaxResults(5);

**HIBERNATE – NATIVE SQL**

Hibernate allows us to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

Syntax:-

**Public SQLQuery createSQLQuery(String sqlcmd)throws HibernateExcpetion;**

Returned results can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

**Scalar Queries**

The most basic SQL query is to get a list of scalars (values) from one or more tables.

Eg:-

SQLQuery qry = session. createSQLQuery("select name, salary from employee";);

qry.setResultTransformer(Criteria.ALIAS\_TO\_ENTITY\_MAP);

List results = qry.list();

**Entity Queries**

SQLQuery qry = session. createSQLQuery(“select \* from employee”);

qry.addEntity(Employee.class);

List results = qry.list();

**HIBERNATE – Set Mappings**

A **Set** is a java collection that does not contain any duplicate element. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most, one null element. So, objects to be added to a set must implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. We can use Set collection in our class when there is no duplicate element required in the collection.

The **<set>** element is new to set the relationship between 2 classes. We can use the **cascade** attribute in the <set> element to tell Hibernate to persist the Child object at the same time as the Parent objects. The **name** attribute is set to the defined **Set** variable in the parent class. For each set variable, we need to define a separate set element in the mapping file.

The **<key>** element is the column in the Child table that holds the foreign key to the parent object.

The **<one-to-many>** element indicates that one Parent object relates to many Child objects and, as such, the Child object must have an parent associated with it. We can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on our requirement.