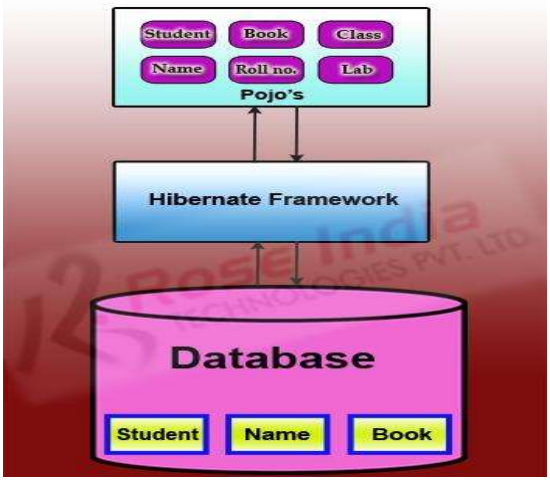
HIBERNATE

Hibernate is one of the most popular ORM tool in Java Programming Language for quickly writing the data access layer for applications. The Hibernate framework was created by Gavin King in the year 2001. Since then Hibernate evolved into enterprise grade Object Relational Mapping tool. Hibernate is high performance, highly scalable, transactional, enterprise grade ORM tool which is being used by developers to develop enterprise applications.

Hibernate increases the productivity of the developers as it generates the persistence related queries on fly. Hibernate framework saves around 95% of envelopers effort in writing the data access layer. It also performs the primary level caching. It can also support secondary-level caching with the help of third party caching framework.

Hibernate is ORM tool : The Hibernate framework is an ORM tool for Java. It is very popular among the developers for creating world class enterprise web and JSE based applications. Hibernate mediates between Java objects and database store. It takes the Java objects from the program and then persists in the database. It takes the instruction from the Java program and translates into a SQL, executes the statements and then returns the result in the Java objects to the Java program.

Following diagram shows how Hibernate works:



Advantages of Hibernate: There are many advantages of using any ORM tool such as Hibernate. First of all it makes programming much cleaner and reduces the development time upto 95%. Here are the advantages of Hibernate framework:

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.
8. Hibernate saves lots of development time
9. POJO class can be directly mapped to the database table.
10. Hibernate automatically generates SQL statement on fly
11. Hibernate works for both JSE and JEE applications

HIBERNATE SUPPORTED DATABASE

Hibernate supports various of databases. Here we are giving some of the Hibernate supported database list and their respective *dialect* i.e., *hibernate.dialect*.

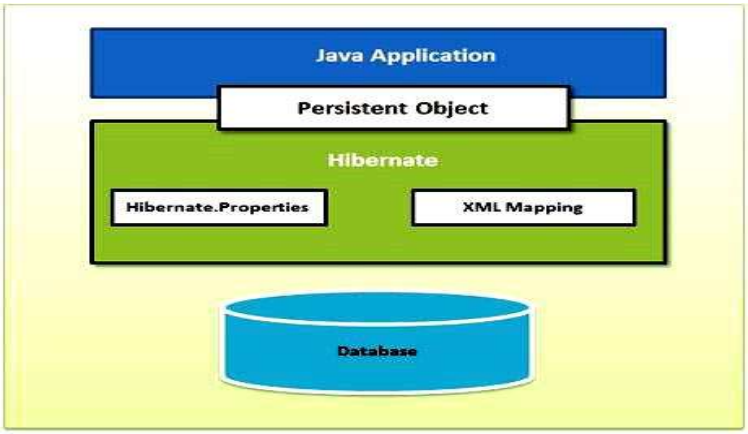
RDBMS	Dialect
MySQL5	org.hibernate.dialect.MySQL5Dialect
MySQL5 with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
H2 Database	org.hibernate.dialect.H2Dialect
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
Sybase	org.hibernate.dialect.SybaseASE15Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

In the above table you can see that the RDBMS names and their respective dialect that will be used in the **hibernate.cfg.xml**. It means which database you will be used in your application to persist the object you will have to use their respective dialect.

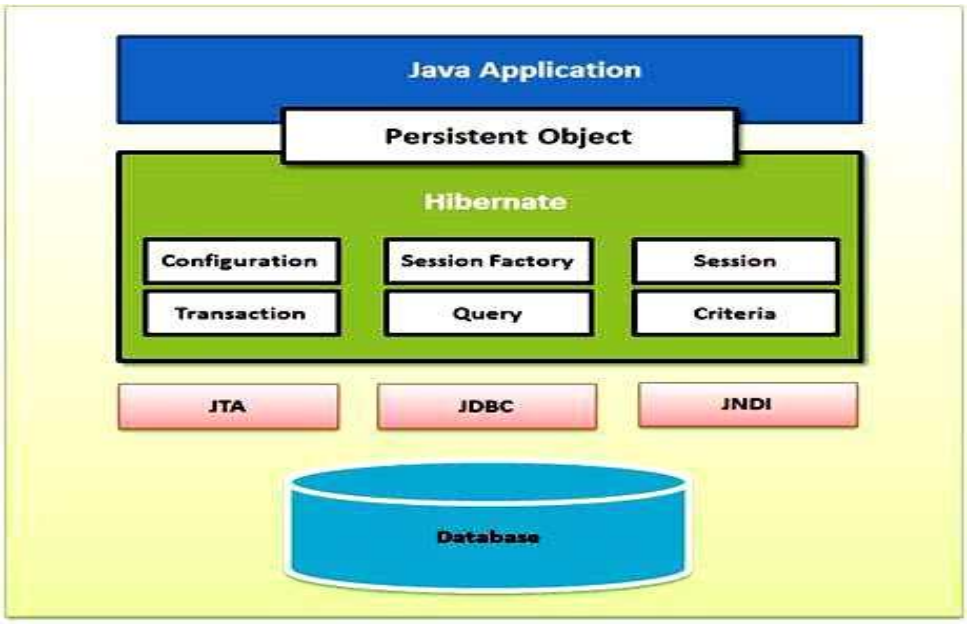
HIBERNATE - ARCHITECTURE

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

The brief description of each of the class objects involved in Hibernate Application Architecture.

1. **Configuration Object:** The Configuration 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.

   The Configuration object provides two keys components −

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

   b. **Class Mapping Setup -** This component creates the connection between the Java classes and database tables.

2. **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 a heavyweight object; it is usually 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.

3. **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.

4. **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).

   This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

5. **Query Object:** 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.

6. **Criteria Object:** Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

Basic Steps for Creating an Hibernate Application

The Hibernate application is developed using the following steps:

1. Writing a POGO class
2. Creating a database table
3. Creating configuration and mapping files
4. Write code to save, retrieve, update, delete objects

1. WRITING POJO CLASS

The first step is to create a Java POJO class. We consider a simple class Book as follows:

```
//Book.java
public class Book {
private int b_id;
private int b_price;
private String b_title;
public Book(){}
public Book(String t, int p) {
this.b_title = t;
this.b_price = p;
}
public int getId() { return b_id; }
public void setId(int id ) { this.b_id = id; }
```

```
public String getTitle() { return b_title; }
public void setTitle(String t ) { this.b_title = t; }
public int getPrice() { return b_price; }
public void setPrice(int p ) { this.b_price = p; }
}
```

This contains three private fields *b_id, b_price* and *b_title* and *get* and *set* methods for these
fields. The *b_id* field works like an index, whereas *b_title* and *b_price* fields store the title and price of a book respectively. It is a good idea to write a class in hibernate as JavaBeans compliant class.

2. CREATING A TABLE:

The next step is to create a database table. Since, MySQL is one of the most popular open-source database systems available today, we shall use it to demonstrate hibernate. We assume that a MySQL instance is running in the host having IP address 172.16.5.81 and the user name and password to access this database are root and nbuser respectively. We also assume that a database test is already created and configured. If it does not exist, create it using the following command at the MySQL command prompt:

mysql> create database test;

Query OK, 1 row affected (0.00 sec)

Now, connect to test database using the following SQL command:

mysql> connect test;

Connection id: 88

Current database: test

Now database is ready. We can create tables in it. For each class, there should exist, a database table which will hold object's state (attribute values). Since, a Book object will have three fields (b_id, b_title and b_price), let us create a table tbl_book having three columns using the following SQL command:

create table tbl_book (no int auto_increment, name varchar(80), pr int, primary key (no));

Here *no, name* and *pr* columns correspond to id, title and price fields of the Book class respectively.

3. WRITING A HIBERNATE APPLICATION

We then write a simple Java hibernate application. This first application simply stores a single

Book object in the table book.

In this application, we first import necessary classes to work with hibernate. Two primary hibernate API packages are *org.hibernate* and *org.hibernate.cfg*. Import classes from these packages:

import org.hibernate.;*

import org.hibernate.cfg.;*

Hibernate – Configuration: Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called *hibernate.properties,* or as an XML file named *hibernate.cfg.xml.*

I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties: Following is the list of important properties, you will be required to configure for a databases in a standalone situation −

1. **hibernate.dialect :** This property makes Hibernate generate the appropriate SQL for the chosen database.
2. **hibernate.connection.driver_class :** The JDBC driver class.
3. **hibernate.connection.url:** The JDBC URL to the database instance.
4. **hibernate.connection.username:** The database username.
5. **hibernate.connection.password:** The database password.
6. **hibernate.connection.pool_size:** Limits the number of connections waiting in the Hibernate database connection pool.
7. **hibernate.connection.autocommit:** Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI, then you would have to configure the following properties :

1. **hibernate.connection.datasource:** The JNDI name defined in the application server context, which you are using for the application.
2. **hibernate.jndi.class:** The InitialContext class for JNDI.
3. **hibernate.jndi.<JNDIpropertyname>:** Passes any JNDI property you like to the JNDI *InitialContext*.
4. **hibernate.jndi.url:** Provides the URL for JNDI.
5. **hibernate.connection.username:** The database username.
6. **hibernate.connection.password:** The database password.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available at http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd.

The configuration information is stored in files called *configuration files*, which may be standard Java properties files or XML files. The object to database table mapping information, on the other hand, may be stored in XML files called *mapping files* or may be specified using annotation. The Configuration object loads information from these files/annotations.

The default constructor of Configuration class always searches a configuration file named *hibernate.properties* (which is an ordinary Java property file) in the current directory. So, create a file with this name and put configuration information there. For MySQL, it looks like this:

#hibernate.properties

hibernate.connection.driver_class com.mysql.jdbc.Driver

hibernate.connection.url jdbc:mysql://172.16.5.81/test

hibernate.connection.username root

hibernate.connection.password nbuser

Alternatively, this information may be put in an XML file (say config.xml) and passed to the configure() method of Configuration object as follows:

cfg.configure("config.xml");

The config.xml file looks like this:

<?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="connection.driver_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://172.16.5.81/test</property>

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

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

</session-factory>

</hibernate-configuration>

This file contains same information as *hibernate.properties* file. An XML configuration file must comply with the hibernate 3 configuration DTD, which is available from *http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd*. We may also use zero argument *configure( )* method as follows:

cfg.configure();

This always expects an XML file with name *hibernate.cfg.xml*. Note that XML configuration file, in contrast to property configuration file, is capable of storing more information such as name of mapping files. The syntax of XML files may be validated against the above mentioned DTD. That is why it is recommended to use XML files as configuration files instead of flat property files.

The Configuration object is aware of database connection properties. Now, we have to specify the object to database table mapping information. A mapping document describes persistent fields and associations. The mapping documents are compiled at application startup time and provide necessary information for a class to the Hibernate framework. They are also used to provide support operations such as creating stub Java source files or generating the database schema etc.
Mapping specified in two ways:

- Using XML files called mapping files or
- Using annotation

Example: Sample file (Book.hbm.xml) that maps Book class to tbl_book table:

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

```
<hibernate-mapping>
<class name="Book" table="tbl_book">
<id name="id" type="int" column="no"/>
<property name="title" column="name" type="string"/>
<property name="price" column="pr" type="int"/>
</class>

</hibernate-mapping>
```

The name of this mapping file is chosen as *[classname].hbm.xml*. A mapping file must have a root element <*hibernate-mapping*>..</*hibernate-mapping*>. The <*class*> element maps a Java class to a database table specified using name and table attributes respectively.

The Configuration object to look for mapping information in the XML file ***Book.hbm.xml*** as follows:

cfg.addResource("Book.hbm.xml");

The name of the mapping file can be specified in the XML configuration file using resource attribute of <mapping> element as follows:

```
<hibernate-configuration>
<session-factory>
<!—Other entries-->
<!-- List of XML mapping files -->
<mapping resource="Book.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

This reads data from configuration and mapping files and finally places them in a high level heavyweight hibernate object, the *SessionFactory* object, an instance of which may be built as follows:

SessionFactory factory = cfg.buildSessionFactory();

We then create a *Session* object, which is primary runtime interface between a Java application and Hibernate:

Session session = factory.openSession();

A Session object is equivalent to *Connection* object of JDBC. The main function of the *Session* object is to allow creating, reading, updating and deleting instances of mapped entity classes. In our application, we save it as a *Book* object. So, we create one such object:

Book b = new Book("Advanced Java Programming", 450);

For a database write (e.g. insert, update, delete) hibernate needs a logical Transaction to be started as follows:

Transaction tx = session.beginTransaction();

Note: For database read operations we do not require any logical transaction in hibernate.

The *Session* object provides useful methods to save, update, delete objects. We use *save*() to save our *Book* object:

session.save(b);

Note: Hibernate, by default, does not automatically commit (autocommit mode false)database transaction.

So, the save() method writes *b* to the database; nothing will persistpermanently in the database until the transaction performs a commit explicitly. Hence, wedo it using commit()method of in Transaction:

```
tx.commit();
```

The entire source code (SaveBook.java) of our application is shown below:

```
//SaveBook.java
import org.hibernate.*;
import org.hibernate.cfg.*;
```

```
public class SaveBook {
public static void main(String[] args) {
Configuration cfg = new Configuration();
cfg.configure("config.xml");
cfg.addResource("Book.hbm.xml");
SessionFactory factory = cfg.buildSessionFactory();
Session session = factory.openSession();
Book b = new Book("Advanced Java Programming", 450);
Transaction tx = session.beginTransaction();
try {
session.save(b);
tx.commit();
}catch (Exception e) {
if (tx!=null) tx.rollback();
}
finally { session.close(); factory.close();}
}
}
```

Annotations used in Hibernate:

We can create the same persistent class using annotation.

1. **@Entity** annotation marks this class as an entity.
2. **@Table** annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name by default.
3. **@Id** annotation marks the identifier for this entity.
4. **@Column** annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default.
5. **@GeneratedValue** This optional annotation is used to specify the primary key generation strategy to use. If none are used, default AUTO will be used.

Import annotation interfaces from javax.persistence package as follows:

```
import javax.persistence.*;
```

The complete source of the modified annotated Book.java is shown below:

```
//Book.java
import javax.persistence.*;
@Entity
@Table(name = "tbl_book")
public class Book {
private int b_id;
private int b_price;
private String b_title;
public Book(){}
public Book(String t, int p) {
this.b_title = t;
this.b_price = p;
}
@Id @GeneratedValue
@Column(name = "no")
public int getId() { return b_id; }
public void setId(int id ) { this.b_id = id; }
@Column(name = "name")
public String getTitle() { return b_title; }
public void setTitle(String t ) { this.b_title = t; }
```

```
@Column(name = "pr")
public int getPrice() { return b_price; }
public void setPrice(int p ) { this.b_price = p; }
}
```

Add annotation class to the hibernate Configuration object:

```
cfg.addAnnotatedClass(Book.class);
```

The following annotation instructs Hibernate to access fields of Book object directly:

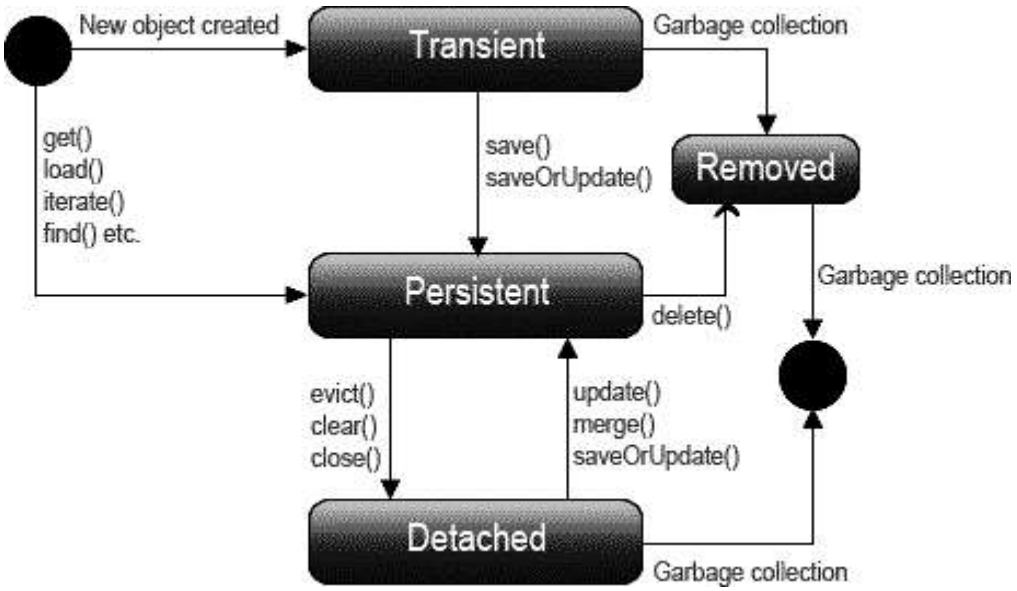
```
//Book.java
import javax.persistence.*;
@Entity
@Table(name = "tbl_book")
public class Book {
@Id @GeneratedValue
@Column(name = "no")
private int b_id;
@Column(name = "pr")
private int b_price;
@Column(name = "name")
private String b_title;
public Book(){}
public Book(String t, int p) {
this.b_title = t;
this.b_price = p;
}
}
```

HIBERNATE OBJECT LIFE CYCLE

Hibernate Life cycle shows how your persistent object managed. Hibernate is ORM based java technology. Hibernate saves, update and delete record in the database table regarding to, value of object as your persistent class is associated with the database table. Hibernate object life cycle contains four states. These are -

1. Transient
2. Persistent
3. Removed
4. Detached

Diagram :



1. **Transient State :** When we create an object of our pojo class ,we can say it is in Transient state. By using new operator, the object is instantiated and at that time it does not enter into persistent but into transient state. At this state it is not related to any database table. At the time of Transient state any modification does not affect database table. If no longer referenced by any other object, it refers to garbage collection.

   Example :

   Employee employee=new Employee(); //employee is a transient object

2. **Persistent State :** When you save your transient object it enter into persistent state. Persistent instance has valid database table row with a primary key identifier .It is instantiated by the persistent manager when you call *save*(). If it is already exist you can retrieve it from the database. This is the state where objects are saved to the database.
   All the changes done in this state, are saved automatically. You can made object persistent by calling against Hibernate session -

   - *session.save()*
   - *session.update()*
   - *session.saveOrUpdate()*
   - *session.lock()*
   - *session.merge()*

   Example :

```
Session session=Session.getCurrentSession();
Employee employee=new Employee() //transient state. Hibernate
is unaware that it exist
session.saveOrUpdate(employee); //persistent state. Hibernate
know the object and will save it to the database
employee.setName("Sonja"); //modification is saved
automatically because the object is in the persistent state
session.getTransaction().commit(); //commit the transaction
```

3. **Detached State :** In this state, the persistent object is still exist after closure of the active session. You can say detached object remains after completion of transaction .It is still represents to the valid row in the database. In detached state, changes done by detached objects are not saved to the database. You cannot detach persistent object explicitly. ***evict***() method is used to detach the object from session cache. ***clear***(), ***close***() methods of session can be used to detach the persistent object.

   For returning in to the persistent state, you can reattach detached object by calling methods -

   - update()
   - merge()
   - saveOrUpdate()
   - lock() - It is reattached but not saved.

   Example :

```
Session session1=Session.getCurrentSession();
```

```
Employee employee=Session1.get(Employee.class, 2);
//retrieve employee having empId 2.employee return
persistent object.
session1.close(); //transition is in detached state.
Hibernate no longer manages the object.
employee.setName("Ron"); //Since it is in detached state so
modification is ignored by Hibernate
Session session2=SessionFactory.getCurrentSession();
//reattach the object to an open session. Return persistent
object and changes is saved to the database.
session2.update(employee);
session.getTransaction.commit(); //commit the transaction
```

4. **Removed State :** When the persistent object is deleted from the database ,it is reached into the removed state.
   session.delete(): At this state java instance exist but any changes made to the object are not saved to the database.
   It is ignored by Hibernate and when it is out of scope, it is assigned to garbage collection.

   Example :

```
Session session=Session.getCurrentSession();
Employee employee=Session.get(Employee.class, 2); //retrieve
employee having empId 2.employee return persistent object.
session.delete(employee); //transition is in removed state.
database record is deleted by Hibernate and no longer manages
the object.
employee.setName("Ron"); //Since it is in removed state so no
modification done by Hibernate
session.getTransaction.commit(); //commit the transaction
```

HIBERNATE - QUERY LANGUAGE

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.

Although you can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.

FROM Clause: You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause −

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

If you need to fully qualify a class name in HQL, just specify the package and class name as follows −

```
String hql = "FROM com.hibernatebook.criteria.Employee";
```

```
Query query = session.createQuery(hql);
List results = query.list();
```

AS Clause: The **AS** clause can be used to assign aliases to the classes in your HQL queries, especially when you have the long queries. For instance, our previous simple example would be the following −

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows −

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

SELECT Clause: The **SELECT** clause provides more control over the result set then the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first_name field of the Employee object −

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

WHERE Clause: If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause −

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

ORDER BY Clause: To sort your HQL query's results, you will need to use the **ORDER BY** clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using ORDER BY clause −

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows −

```
String hql = "FROM Employee E WHERE E.id > 10 " +
             "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY Clause: This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause −

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
             "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

Using Named Parameters: Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters –

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

UPDATE Clause: Bulk updates are new to HQL with Hibernate 3, and delete work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The **UPDATE** clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause –

```
String hql = "UPDATE Employee set salary = :salary "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE Clause: The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause –

```
String hql = "DELETE FROM Employee "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT Clause: HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause –

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

AGGREGATE METHODS: HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions –

1 **avg(property name):**The average of a property's value

2 **count(property name or *):** The number of times a property occurs in the results

3 **max(property name):** The maximum value of the property values

4 **min(property name):** The minimum value of the property values

5 **sum(property name):** The sum total of the property values

The **distinct** keyword only counts the unique values in the row set. The following query will return only unique count −

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

Pagination using Query: There are two methods of the Query interface for pagination.

1. **Query setFirstResult(int startPosition):** This method takes an integer that represents the first row in your result set, starting with row 0.
2. **Query setMaxResults(int maxResult):** This method tells Hibernate to retrieve a fixed number **maxResults** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example, which you can extend to fetch 10 rows at a time:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

NAMED QUERIES

Hibernate provides a technique called **named queries** that lets us separate queries from the coding section of the application. Developers place queries in mapping xml file (.hbm files) or annotated class and assign unique names to them. A Java application can refer to those queries by their name. The following are some advantages of named queries:

- Since, queries are no longer scattered in Java code, it greatly helps in code cleanup.
- It is possible to use the same query multiple times without writing the same query multiple times.
- Their syntax is checked when the session factory is created, making the application fail safe in case of an error.
- If queries are placed in the mapping file, no recompilation is required in case of query modification.

Since most of the cases queries are cached, reloading of session factory is required which may result in server restart up. Named queries sometimes make debugging difficult as we have to locate the actual query definition being executed and understand that as well. Hibernate supports both HQL as well as native SQL named queries.

Defining Named Queries: The named queries may be defined in the mapping file or in an annotated class.

Using Mapping File: We define an HQL named query using <query> tag in the mapping file as follows:

```
<hibernate-mapping default-access="property">
…
<query name="find_All_Books_HQL">
from Book
</query>
…
</hibernate-mapping>
```

If there is a <*class*> element, place <*query*> element before it. The <query> tag has a

application. The query itself is written within the <*query*> tag. Since a query may contain some special characters (such as <, > etc), it is usually placed within CDATA section as follows:

```
<query name="find_Books_price_less_than_300_HQL">
<![CDATA[from Book where price < 300]]>
</query>
```

An SQL named query is defined using <sql-query> tag as follows:

```
<sql-query name="find_All_Books_SQL">
<return class="Book"/>
select * from tbl_book
</sql-query>
```

The <*return*> tag specifies entity object to be returned by the SQL query. The CDATA section is used to embed query as it may contain special characters such as <, > etc. as follows:

```
<sql-query name="find_Books_price_less_than_300_SQL">
<return class="Book"/>
<![CDATA[select * from tbl_book where pr < 300]]>
</sql-query>
```

Using Annotation: Named HQL queries may be defined in an annotated class using *@NamedQueries* annotation as follows:

```
@NamedQueries({
@NamedQuery(name="find_All_Books_HQL", query="from Book")
})
```

The *@NamedQueries* annotation contains a comma-separated list of queries, each of which is specified by *@NamedQuery* annotation.

@NamedQuery annotation has two important attributes: name and query. The name specifies the name of the query by which it will be referred and the query specifies the actual HQL query string to be executed in database. Multiple queries are defined as follows:

```
@NamedQueries({
@NamedQuery(name="find_All_Books_HQL", query="from Book"),
@NamedQuery(name="find_Books_price_less_than_300_HQL",
query="from Book where price < 300")
})
```

SQL named queries are defined using @NamedNativeQueries and @NamedNativeQuery annotations.

```
@NamedNativeQueries({
@NamedNativeQuery(name="find_All_Books_SQL", query = "select *
from tbl_book", resultClass=Book.class ), })
```

The *resultClass* attribute specifies entity to be returned by the SQL query. Multiple queries are defined as follows:

```
@NamedNativeQueries({
```

```
@NamedNativeQuery(name="find_All_Books_SQL", query = "select *
from tbl_book", resultClass=Book.class),

@NamedNativeQuery(name="find_Books_price_less_than_300_SQL",
query = "select * from tbl_book where pr < 300",
resultClass=Book.class)
})
```

The names of queries must be unique in XML mapping files or annotations. Place all named queries in a separate file and include it in the Java code or in the configuration file. Here is an example of named query file (NQ.hbm.xml):

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping SYSTEM
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<query name="find_All_Books_HQL"><![CDATA[from Book]]</query>
<sql-query name="find_All_Books_SQL">
<return class="Book"/>
<![CDATA[select * from tbl_book]]
</sql-query>
</hibernate-mapping>
```

To include this file in Java code, use addResource() of Configuration object as follows:

```
Configuration cfg = new Configuration();
//add configuration file
cfg.addResource("NQ.hbm.xml");
```

Alternatively, include the mapping resource in the configuration file (say config.xml) using

```
<mapping resource="NQ.hbm.xml"/> and add the configuration
file as follows:
cfg.configure("config.xml");
```

Calling Named Queries: Once a query is defined, it's name may be used in *getNamedQuery()* method to create a Query object as follows:

```
Query query = session.getNamedQuery("find_All_Books_HQL");
```

Execute the query calling its list() method:

```
List<Book> books = query.list();
```

This list can then be iterated to obtain information about all Book objects as follows:

```
for (Book b : books)
System.out.println(b.getId() +" "+b.getTitle()+" "+ b.getPrice());
```

Here is the complete source code (NamedQueryTest.java) of Java program that uses named query stored in ***NQ.hbm.xml*** file:

```
//NamedQueryTest.java
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;
public class NamedQueryTest {
public static void main(String NamedQueryTest.java
import org.hibernate.*;
```

```
import java.util.*;
public class NamedQueryTest {
public static void main(String[] args) {
Configuration cfg = new Configuration();
cfg.configure("config.xml");
cfg.addResource("Book.hbm.xml");
cfg.addResource("NQ.hbm.xml");
SessionFactory factory = cfg.buildSessionFactory();
Session session = factory.openSession();
Transaction tx=session.beginTransaction();
try {
Query query = session.getNamedQuery("find_All_Books_HQL");
List<Book> books = query.list();
for (Book b : books)
System.out.println(b.getId() +" "+b.getTitle()+" "+ b.getPrice());
tx.commit();
}catch (Exception e) {
e.printStackTrace();
if (tx!=null) tx.rollback();
}
finally { session.close();factory.close();}
}
}
```

GENERATING DDL

Hibernate also provides useful classes to generate DDL. The generated schema may be printed to the console, stored in a file for later use or even exported to a database. The *org.hibernate.tool.hbm2ddl.SchemaExport* class can be used for these purposes. The *SchemaExport* object is created from a given Configuration object as follows:

```
SchemaExport schemaExport = new SchemaExport(cfg);
The schema is then created using its create() method:
boolean print = true, export = true;
schemaExport.create(print, true);
```

This runs the schema creation script. The *drop()* method is automatically executed before running the creation script. The variables print and export indicate that the schema has to be printed to the console and exported to the database. The following is the complete source code of the program:

```
//GenerateDDL.java
import org.hibernate.cfg.*;
import org.hibernate.tool.hbm2ddl.*;
public class GenerateDDL {
public static void main(String[] args) {
Configuration cfg = new Configuration();
cfg.configure("config.xml");
cfg.addResource("Book.hbm.xml");
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.setDelimiter(";");
schemaExport.setOutputFile("out.sql");
boolean print = true, export = true;
schemaExport.create(print, true);
}
}
```

The program also stores the schema in a file out.sql containing the following SQL statements:

```
drop table if exists tbl_book;
create table tbl_book (no integer not null auto_increment,
name varchar(255), pr integer, primary key (no) );
```