**FRAMEWORK**

->a software framework is set of source code or library which provide specific functionality to application.

->software framework consist of frozen spot & hot spot.

**FROZEN SPOT**

->it defines the overall architecture of a framework i.e. its basic component & relationship between them.these remain unchanged(frozen).

**HOT SPOT**

->it represent those parts where the programmer using the framework add their own code to add the functionality ,specific to their own project.

->hibernate is collection of jar file.

->few example of framework log4j,hibernate,spring.

**MAJOR ADVANTAGE OF FRAMEWORK**

1)using code which has already bean built,tested & used by other programmer increases reliability and reduces the programming time.in a co-operation this code reuse effectively save money.

2)framework often used based industries practices & design pattern hence our application indirectly inherits these features.

**HIBERNATE**

->hibernate is an open source DB independent,object relational mapping(ORM) tool for java that help developer to write data handling logic without caring about DB.

->hibernate significantly reduce development time & make its easier to build robust,high performance J2EE application.

->hibernate takes care of mapping java class to DB table and from java data type to SQL type.

**DISADVANTAGES WITH JDBC**

1)even though JDBC is not dependent of DB but since it uses native SQL query directly into it.it is indirectly become DB dependent.

2)we have to manually take care of mapping the query execution result(the data present in resultset ) to java beans.

3)if DB design changes then it will be too expensive to identified changes where changes are required.

4)we have to manually manage the DB connection(like opening and closing the DB connection or may be via connection pool etc).

5)we need to take care of manual closing of all JDBC object.

**ADVANTAGE WITH HIBERNATE**

1)hibernate make our application independent of DB.later if our DB changes from the application code point of view,we need not worry much.we just need to worry about making proper configuration.

2)developer need not to worry about converting the resultset data into java bean with hibernate ,it’s the default behavior.

3)developer need to have less or no SQL knowledge.

4)with hibernate we need not to worry about performance oriented query because hibernate internally take care of these.

5)maintenance of the application is easier,its easier to debug the code & easier to write error free code.

**SETTING OF HIBERNATE**

**STEP 1**:-configure the hibernate code and support jar file to project classpath.

HIBERNATE CODE JAR FILE

9 jar file

SUPPORT JAR FILE

Slt4j-simple 1.6.1

Mysql-connector-jar

**STEP 2**:-create a hibernate.cfg.xml configuration file to describe how to access the DB.

**STEP 3:-**configure the appropriate SQLdialect for the underlying DB in the configuration file.

**NECESSARY STEPS TO WORK WITH HIBERNATE**

**Configuration Steps**

->create the javabean which replicate the DB table.

->create the mapping xml file,which map the bean properties to DB table field.

->add the mapping file to hibernate configuration file.

**Program Steps**

->load the configuration file.

->create the session factory.

->open the session.

->operate with DB.

->flush the session.

->close the session.

**ABOUT CONFIGURATION FILE**

->hibernate is highly configurable through two types of configuration.

1)hibernate configuration file

2)mapping description file.

1)**Hibernate Configuration File**

->on startup hibernate consult these file for it operating properties,such as DB connection URL,DB username & password,DB dialect,datasource,jndi.class,**<JNDIpropertyname>** ,**jndi.url** and mapping file location.

->this file should have the name **hibernate.cfg.xml** or **hibernate.properties.**

->this file should be there in project classpath(under source(src) folder).

->if both files are present in the class path then hibernate.cfg.xml file overrides the setting found in the hibernate.properties file.

**<hibernate-configuration>**

**<session-factory>**

**<property name="hibernate.dialect">**

**org.hibernate.dialect.MySQLDialect**

**</property>**

**<property name="hibernate.connection.driver\_class">**

**com.mysql.jdbc.Driver**

**</property>**

**<!-- Assume test is the database name -->**

**<property name="hibernate.connection.url">**

**jdbc:mysql://localhost/test**

**</property>**

**<property name="hibernate.connection.username">**

**root**

**</property>**

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

**root123**

**</property>**

**<!-- List of XML mapping files -->**

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

**</session-factory>**

**</hibernate-configuration>**

2)**Mapping Description File**

**-**>mapping description file instruct the hibernate how to map data between specific java class & DB table.

->usually filename of the mapping description file will have the following structure.

**<bean-name>.hbm.xml**

But this is not mandatory.it should be a xml file and can have any name(abc.xml).

->we should not use some java bean is more than one mapping file.

->javabean configure in the mapping file should have the configure number of property init.however it can also hava extra property.hibernate doesnot throw any exception in this case.

->Db table & mapping file should be in sink with regard to DB column file and java bean property file.in other words java bean should be a exact replicate of DB table.

HIBERNATE

Hibernate.cfg.xml

Jar

Java

Mapping.xml

Test Hibernate

DB

JDBC

DIALECT

HIBERNATE

1)MySQL

2)Oracle

**<hibernate-mapping>**

**<class name="Employee" table="EMPLOYEE">**

**<meta attribute="class-description">**

**This class contains the employee detail.**

**</meta>**

**<id name="id" type="int" column="id">**

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

**</id>**

**<property name="firstName" column="first\_name" type="string"/>**

**<property name="lastName" column="last\_name" type="string"/>**

**<property name="salary" column="salary" type="int"/>**

**</class>**

**</hibernate-mapping>**

**HIBERNATE DIALECT**

**->**hibernate is DB independent ORM tool so whatever DB we use in our application,we need to set dialect related to that DB in hibernate configuration file.

->Dialect is a “helper” for hibernate to communicate with DB in its language.

->hibernate makes use of dialect file to connect hibernate specific DB call to native DB SQL call.

**JAVABEAN(POJO-PLAIN OLD JAVA OBJECT)**

->javabean is a plain java class which has

1)public default constructor

2)private variable

3)getter & setter method to access them

4)should not haveany computational logic within that class.

->in case of hibernate its better to override toString() method of object class.

**SESSION FACTORY**

->session factory contain all the hibernate mapping information and it is responsible for creation and maintenance of hibernate session.

->SessionFactory is an interface which create session and its thread safe so that many threads can access it concurrently.

->SessionFactory will be build only at the time of its stratup.

->usually an application will have single session factory.it is immutable.the behavior of SessionFactory is controlled by property supplied at configuration time.

**HIBERNATE SESSION**

->Session is the primary interface use to interact with DB.

->session is a light weight and non thread safe object.that represent a single unit of work with DB.

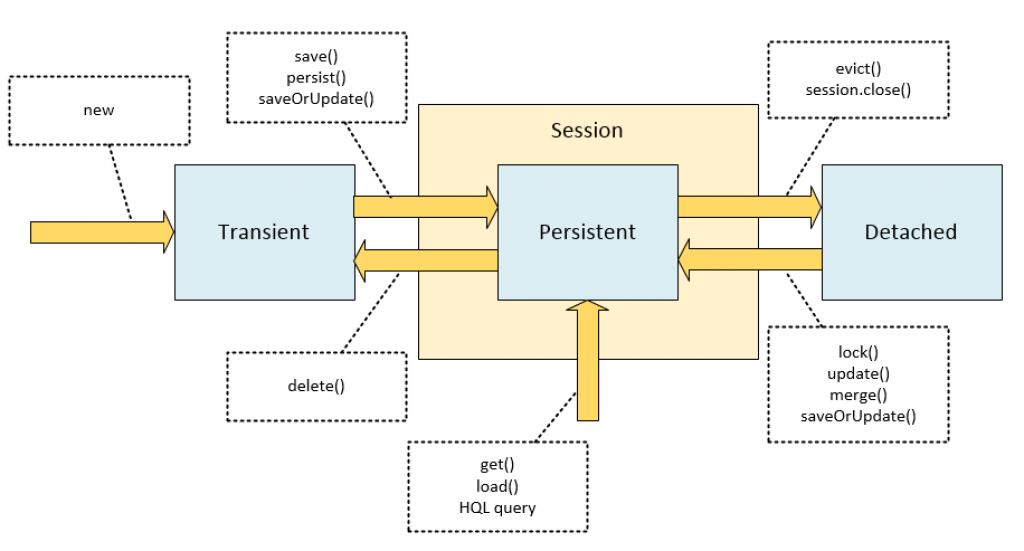
->session are open by SessionFactory and then are closed when all work completes.

->a session obtain a DB connection lazzly.

->the life cycle of session is bounded by beginning and end of logical transaction.

->by calling flush method on session object,we force the hibernate to execute the SQL command on DB.

### **States of Entity Instances**



**Save() and persist())**

1. When we are giving the primary key(using assigned generator class) then no difference.
2. When hibernate will assign the primary key(other than assigned generator class EX. Increment) then save method will return that primary key value but persist() method doesn’t return any value.
3. long s = session.save(k);

**SaveOrUpdate()**

Calls either save() or update() on the basis of identifier exists or not. e.g if identifier exists, update() will be called or else save() will be called.

**Update()**

 Suppose we are dealing with any employee object in the same session then we should use updaQte() or saveOrUpdate() method.

if you are sure that the session does not contains an already persistent instance with the same identifier,then use update to save the data in hibernate

**Merge()**

Suppose we are creating a session and load an employee object. Now object in session cache. If we close the session at this point and we edit state of object and tried to save using update() it will throw exception. To make object persistent we need to open another session. Now we load same object again in current session. So if we want to update present object with previous object changes we have to use merge() method. Merge method will merge changes of both states of object and will save in database.

if you want to save your modifications at any time with out knowing about the state of an session, then use merge() in hibernate.

**refresh()**

It is used to synchronize database data with session data.

**TRANSIENT**

A new instance of a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.

**PERSISTENT**

You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.

**DETACHED**

Once we close the Hibernate Session, the persistent instance will become a detached instance.

**PERSISTENT CLASS**

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate.

**HIBERNATE ARCHITECTURE**

**APPLICATION BUSINESS LOGIC**

POJO HIBERNATE

DB

JDBC API’S

Database dialect

Hibernate api

Mapping.hbm.xml

Hibernate.cfg.xml

**HANDLING TRANSACTION IN HIBERNATE**

->transaction handling in hibernate is similar to handling transaction in JDBC.

->we begin the transaction by invoking beginTransaction() method on session object.

->once done either commit the transaction or rollback the transaction by invoking commit() and rollback() method on transaction object respectively.

**Session session = factory.openSession();**

**Transaction tx = null;**

**try {**

**tx = session.beginTransaction();**

**// do some work**

**...**

**tx.commit();**

**}**

**catch (Exception e) {**

**if (tx!=null) tx.rollback();**

**e.printStackTrace();**

**}finally {**

**session.close();**

**}**

**HANDLING THE DB TABLE DEFAULT VALUE IN HIBERNATE**

->if a table has any default values then declares default values in javabean.

**FIND BY PRIMARY KEY**

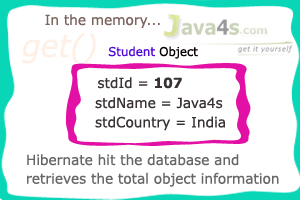
->in enterprise application an efficient such mechanism is highly needed.hibernate provide a set of different technique to search a set of persistence object.

->find by primary key is a simple mechanism to find an object by primary key without using a SQL query.

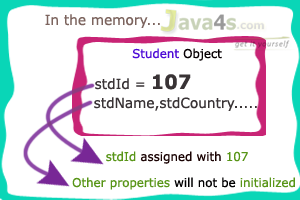
**DIFFERENCE BETWEEN GET & LOAD**

|  |  |
| --- | --- |
| **GET** | **LOAD** |
| 1)it returns null object if the record doesnot exist with a given primary key | 1)it throws object not found exception,if the record doesn’t exist with the given primary key. |
| 2)get method hits the database as soon as it is called ,so this method will always trigger a database hit. | 2)the load method only hits the database when a particular field of the entity is accessed.so if we use the load method to retrieve an entity but we never actually access any of the field of that entity.we never hit the databases. |
| 3)if we don’t know for sure that whatever record exist in DB or not with a given primary key then we can use the get method and do a null check on the instance that get return from the get method call. | 3)the load method may be the method of choice if know and absolutely sure that the record we are searching existing DB with the given primary key. |
| 4) Eager fetching. | 4) Lazy fetching. |
| 5) Fast if record exists. | 5) Slow if record exists. |

* When you call session.get() method, it will hit the database immediately and returns the original object.



* When you call session.load() method, it will always return a “proxy” object,  whats the meaning of proxy object ?
* Proxy means, hibernate will prepare some fake object with given identifier value in the memory without hitting the database, for example if we call session.load(Student.class,new Integer(107));  > hibernate will create one fake Student object [row] in the memory with id 107, but remaining properties of Student class will not even be initialized, observe this graphical representation…



1. It will hit the database only when we try to retrieve the other properties of Student object i mean stdName, stdCountry.  If we call s2.getStdName() then hibernate will hit the database and search the row with student id 107 and retrieve the values, if object [row] not found in the database it will throws ObjectNotFoundException.

Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

**EXAMPLE**

**1)POJO CLASS**

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

public Employee() {}

public Employee(String fname, String lname, int salary) {

this.firstName = fname;

this.lastName = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

}

## 2) Create Database Tables

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

## 3) Create Mapping Configuration File

<?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="Employee" table="EMPLOYEE">

<meta attribute="class-description">

This class contains the employee detail.

</meta>

<id name="id" type="int" column="id">

<generator class="native"/>

</id>

<property name="firstName" column="first\_name" type="string"/>

<property name="lastName" column="last\_name" type="string"/>

<property name="salary" column="salary" type="int"/>

</class>

</hibernate-mapping>

## 4) Create Application Class

import java.util.List;

import java.util.Date;

import java.util.Iterator;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class ManageEmployee {

private static SessionFactory factory;

public static void main(String[] args) {

try{

factory = new Configuration().configure().buildSessionFactory();

}catch (Throwable ex) {

System.err.println("Failed to create sessionFactory object." + ex);

throw new ExceptionInInitializerError(ex);

}

ManageEmployee ME = new ManageEmployee();

/\* Add few employee records in database \*/

Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);

Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);

Integer empID3 = ME.addEmployee("John", "Paul", 10000);

/\* List down all the employees \*/

ME.listEmployees();

/\* Update employee's records \*/

ME.updateEmployee(empID1, 5000);

/\* Delete an employee from the database \*/

ME.deleteEmployee(empID2);

/\* List down new list of the employees \*/

ME.listEmployees();

}

/\* Method to CREATE an employee in the database \*/

public Integer addEmployee(String fname, String lname, int salary){

Session session = factory.openSession();

Transaction tx = null;

Integer employeeID = null;

try{

tx = session.beginTransaction();

Employee employee = new Employee(fname, lname, salary);

employeeID = (Integer) session.save(employee);

tx.commit();

}catch (HibernateException e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

return employeeID;

}

/\* Method to READ all the employees \*/

public void listEmployees( ){

Session session = factory.openSession();

Transaction tx = null;

try{

tx = session.beginTransaction();

List employees = session.createQuery("FROM Employee").list();

for (Iterator iterator =

employees.iterator(); iterator.hasNext();){

Employee employee = (Employee) iterator.next();

System.out.print("First Name: " + employee.getFirstName());

System.out.print(" Last Name: " + employee.getLastName());

System.out.println(" Salary: " + employee.getSalary());

}

tx.commit();

}catch (HibernateException e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

}

/\* Method to UPDATE salary for an employee \*/

public void updateEmployee(Integer EmployeeID, int salary ){

Session session = factory.openSession();

Transaction tx = null;

try{

tx = session.beginTransaction();

Employee employee =

(Employee)session.get(Employee.class, EmployeeID);

employee.setSalary( salary );

session.update(employee);

tx.commit();

}catch (HibernateException e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

}

/\* Method to DELETE an employee from the records \*/

public void deleteEmployee(Integer EmployeeID){

Session session = factory.openSession();

Transaction tx = null;

try{

tx = session.beginTransaction();

Employee employee =

(Employee)session.get(Employee.class, EmployeeID);

session.delete(employee);

tx.commit();

}catch (HibernateException e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

}

}

## EXAMPLE WITH ANNOTATION

## WRITING ONLY THE DIFFERENCE

**1)POJO CLASS**

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id @GeneratedValue

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

}

## 2) CREATE APPLICATION CLASS

public class ManageEmployee {

private static SessionFactory factory;

public static void main(String[] args) {

try{

factory = new AnnotationConfiguration().

configure().

addAnnotatedClass(Employee.class).

buildSessionFactory();

## }

**RELATIONSHIP(association exist in any DB)**

1)one to one

2)one to many

3)many to one

4)many to many

**MANAGING ASSOCIATION IN HIBERNATE**

**->**association is a relationship of one javabean to another javabean.it also known as relationship in DB terminology.

->usually DB design involves tables to have relationship with one to another in RDBMS table can have following relationship.

1)one to one

2)one to many

3)many to one

4)many to many

**ONE TO ONE RELATIONSHIP IN HIBERNATE**

->one-to-one relationship occurs when there is exactly one record in 1st table correspond to exactly one record in the related table.

Table 2

ID(PK)

COLUMN 1

COLUMN2

Table 1

ID(PK)

COLUMN1

COLUMN2

1 1

**CASCADE**

->cascade is a very convenient feature to manage the state of the child object automatically.cascade attribute transfers operation done on one object onto its related child object.

->however this feature comes with twice.if we don’t use it twicely,it will generate unnecessary side effect and slow down the application performance.

|  |  |
| --- | --- |
| **CASCADE OPTIONS** | **DESCRIPTION** |
| 1)cascade=”none” | It’s the default option which tells hibernate to ignore the association. |
| 2)cascade=”all” | All operation like save,delete,update at parent object will be affected to child object as well. |
| 3)cascade=”save” | Operation like save at parent object will be affected to child object as well. |
| 4)cascade=”update” | Operation like update at parent object will be affected to child object as well. |
| 5)cascade=”delete” | Operation like delete at parent object will be affected to child object as well. |
| 6)cascade=”save-update” | Operation like save & update at parent object will be affected to child object as well. |
| 7)cascade=”delete-orphan” | An orphan record means,it is a record in child table but doesn’t association with parent.then hibernate will delete any record which doesn’t have association with parent. |
| 8)cascade=”all-delete-orphan” | The operation as same as cascade=”all” but in addition hibernate deletes the orphan records. |

**MANY-TO-ONE RELATIONSHIP IN HIBERNATE**

->many-to-one relationship occurs when multiple records of one table corresponds to exactly one record in another table.

Table 2

anotherID(PK)

COLUMN 1

COLUMN2

Table 1

ID(PK)

anotherID

COLUMN1

COLUMN2

1

N

**FETCHING STRATEGIES**

1)fetching strategies are highly flexible and are very important tweak to optimize to hibernate query.but if it is used it in wrong place then it will be total disaster.

2)**fetch=”select”-**it’s a default.this will lazily load all the collection and entities.

3) **fetch=”join”-**it will always load all the collection and entities.

**ONE-TO-MANY MAPPING**

->one-to-many mapping occurs when a single record of one table corresponds to multiple records in another table.

Table 2

anotherID(PK)

COLUMN 1

COLUMN2

Table 1

ID(PK)

anotherID

COLUMN1

COLUMN2

N

1

**MANY-TO-MANY MAPPING**

->in case of many-to-many relationship many record of one table will have corresponding many records in another table.

Table 2

ID2(PK)

COLUMN 1

COLUMN2

Table 1

ID1(PK)

anotherID

COLUMN1

COLUMN2

1 N N N 1

ID1

ID2 PK

**SUMMARY 1**

USERS

1

STUDENT\_INTERVIEW

INTERVIEW\_DETAILS

PHONE NUMBER

COURSE

STUDENTS

1 N

1 N

1:N N:1

**INVERSE KEYWORD**

1)in hibernate only the relationship owner should maintain the relationship and inverse keyword tells which side is the owner to maintain the relationship.

2)the inverse keyword is used with one-to-many and many-to-many relationship.many-to-one and one-to-one doesn’t have inverse keyword.

3)inverse=”true” means the current entity is the relationship owner and inverse=”false”(its default) means,its not the owner.

**LAZY KEYWORD**

1)lazy keyword decides whether to load child object while loading the parent object.

2)lazy=”true”(its default) means not to load child object.this make sure that child objects are not loaded unless they are explicitly invoked in the application by calling the get() methd on parent.

3)lazy=”false” means hibernate will load the child object when parent is loaded from the database.

4)if we try to access the child objects lazily(lazy=”true”) after closing the session then we will get lazy initialization exception.

**DIFF BETWEEN LAZY AND FETCH**

1)lazy attribute tells hibernate when to get the children,fetch attribute tells hibernate how get the children.

2)fetch=”select” and lazy=”true” both means same.

**NOT-NULL KEYWORD**

1)not-null=”true” specifies that column can’t hold null value.not-null=”false” specifies that column can hold null value.

**Inheritance in Hibernate**

1. Three types of inheritance in hibernate.

**Table per class** : All parent and child data will store into one table. This table will contain one extra column that is discriminator. that is used to identify which child data this is. Disadvantage is column must contain null value.

**Table per subclass** : To resolve above issue(column must not contain null value) we have to create multiple table for each parent and child object. Disadvantages : If we have to fetch the data for particular employee then we have to write complex query.

**Table per concreteclass :** Here by default parent class will be abstract class and we have to create table for only concrete(child) classes, not for abstract(parent) classes.

**N+1 problem in Hibernate**

1. Suppose we have one to many relationship between department and employee. So we have to display all the employee of each department.

"select \* from Department";

"select \* from Employee where deptId=?";

1. Here one select statement will fire for deptid and suppose we have n dept then N number of query will fire to get the employee of each dept. So this is called N+1 problem.
2. To resolve above issue we have to join both the table and display employee using only one select query.

"from Department d join fetch d. Employee e";

**HQL(HIBERNATE QUERY LANGUAGE)**

1)operates on java beans,DB independent.

2)SQL-operates on DB tables,DB dependent.

1)hibernate provides a query language similar to standard SQL to perform operations on hibernate objects.

2)the advantages of using a HQL is that,SQL varies as different database or adopted to implement different solution but HQL remains same as long as we are within the domain or hibernate;no matter which database are using.

3)HQL is case insensitive except the java class and attributes name.hence ‘from’ and ‘FROM’ are same but ‘Students’ and ‘students’ are not.

4)we can also use direct package name in HQL.

String sql1=”from com.jspider.bean.Students”;

EXAMPLE-

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();

**HIBERNATE CRITERIA QUERY**

1)criteria query works on java object.we can add more than one restriction to criteria queries.

EXAMPLE-

Cr

iteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000

cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara

cr.add(Restrictions.like("firstName", "zara%"));

List results = cr.list();

**HIBERNATE NATIVE SQL**

our application will create a native SQL query from the session with the **createSQLQuery()** method on the Session interface.

String sql = "SELECT \* FROM EMPLOYEE WHERE id = :employee\_id";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

query.setParameter("employee\_id", 10);

List results = query.list();

**DDL Generation**

1) Hibernate provides a tool toautomatically generate databaseobjects based on a domain model, or a

domain model based on an alreadyexisting database.

2) hbm2ddl

3)Used through ANT tasks or with Hibernate configuration.

**<!-- in the Hiberante.cfg.xml file -->**

**<session-factory>**

**<property name="hibernate.hbm2ddl.auto">**

**create|create-drop**

**</property>**

**...**

**</sessionFactory>**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**// programmatically**

**Configuration cfg =**

**new Configuration().configure();**

**SchemaUpdate schemaUpdate =**

**new SchemaUpdate(cfg);**

**schemaUpdate.execute();**

**BATCH PROCESSING**

1. When we are executing operations with large data sets then we have to run directly in the database not in memory.

**HIBERNATE BATCH UPDATE**

**Query q =**

**session.createQuery(**

**"update [versioned] Account set balance=**

**(balance + (balance\*interestRate))**

**where accountType='SAVINGS' ");**

**// return number of objects updated**

**int updatedItems = q.executeUpdate();**

**HIBERNATE BATCH DELETE**

**// Provide the monthly interest**

**// to savings accounts**

**Query q =**

**session.createQuery(**

**"delete from Account");**

**// return number of objects deleted**

**// across all subclasses**

**int deletedItems = q.executeUpdate();**

**HIBERNATE BATCH INSERT**

**// Archive all existing accounts**

**Query q =**

**session.createQuery(**

**"insert into ArchivedAccount(**

**accountId, creationDate, balance)**

**select**

**a.accountId, a.creationDate, a.balance**

**from Account a");**

**int createdObjects = q.executeUpdate();**

**DATA FILTERING**

**1)it’s a process through we can limit the amount of data visible without changing query parameters.**

**2)used for security purpose.**

**HOW TO CREATE DATA FILTERS**

**1)Define the filter within the mapping file of the targeted entity.**

**2)Apply the filter in class or collection by indicating it within the <class> or <collection-type> tags.**

**3)After obtaining a session enable or disable the appropriate filter with parameters.**

**Example-**

**<class name="courses.hibernate.vo.Account"table="ACCOUNT">**

**<id name="accountId" column="ACCOUNT\_ID">**

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

**</id>**

**...**

**<filter name="creationDateFilter"**

**condition="CREATION\_DATE >:asOfDate"/>**

**</class>**

**<filter-def name="creationDateFilter">**

**<filter-param name="asOfDate" type="date" />**

**</filter-def>**

**Session session = HibernateUtil.getSessionFactory().getCurrentSession();**

**session.beginTransaction();**

**session.enableFilter("creationDateFilter")**

**.setParameter("asOfDate",**

**new Date(2008,12,08));**

**List accounts = accountService.getAccounts();**

**Assert.assertEquals(2, accounts.size());**

**session.disableFilter("creationDateFilter");**

**accounts = accountService.getAccounts();**

**Assert.assertEquals(5, accounts.size());**

# Hibernate named query

* **Use to segregate the query at one place instead writing query at every place in application.**
* **It allows to write query at entity level.**

**HIBERNATE INTERCEPTORS**

1)suppose we want to perform some task when state of the object changes. For that we can use interceptors.

2)Hibernate has a powerful feature called ‘**interceptor**‘ to intercept or hook different kind of Hibernate events, like database CRUD operation. In this article, i will demonstrate how to implement an application audit log feature by using Hibernate interceptor, it will log all the Hibernate save, update or delete operations into a database table named ‘**auditlog**‘.

**STEPS TO CREATE INTERCEPTORS**

1).Extend the EmptyInterceptor class

2) Implement the desired callback methods

onDelete()

onSave (…)

onDelete(…)

onFlushDirty()

preFlush()

postFlush(…)

etc...

3) Configure the interceptor use

->Either during sessionfactory creation

configuration.setInterceptor(Interceptor)

* After obtaining a session

session = HibernateUtil.getSessionFactory().openSession(interceptor);

**HIBERNATE EVENT SYSTEM**

1. Suppose we are doing some CRUD operation and send some notification after the change then we can implement the event interface.
2. This is the concept of listeners and event.
3. There are multiple types of events that is :

Create(pre & post)

Read(pre & post)

Update(pre & post)

Delete(pre & post)

**CALLBACK INTERFACES**

1)used to receive notification when some operations occurred like loaded,saved or delete.

2)No need to implements callbacks in hibernate but they are useful to implement generic functionality.

**public**List getMostRecent(**final int**count) {

**return**(List) getHibernateTemplate().execute(**new**HibernateCallback() {

**public**Object doInHibernate(Session session) **throws**HibernateException, SQLException {

        Query query = session.createQuery("from Entry");

        query.setMaxResults(count);

**return**query.list();

**DIFF BETWEEN EVENTS AND CALLBACK**

**Events** - Think of a Server (Employee) and Client (Boss).One Employee can have many Bosses. The Employee Raises the event, when he finishes the task, and the Bosses may decide to listen to the Employee event or not.The employee is the publisher and the bosses are subscriber.

**Callback** - The Boss specifically asked the employee to do a task and at the end of task done, the Boss wants to be notified. The employee will make sure that when the task is done, he notifies only the Boss that requested, not necessary all the Bosses. The employee will not notify the Boss, if the partial job is done. It will be only after all the task is done.Only one boss requested the info, and employee only posted the reply to one boss.

**Hibernate - Caching**

1. Used to optimize the performance of any application.
2. Caching is nothing but some buffer where a record is stored when first time retrieved from the database.
3. Second time if we need the same record then Hibernate does not access the database but reads from the **cache**.
4. This type of adjustment decreases the database hits. Accessing cache is much faster than accessing the database.

**There are two types of cache mechanism**

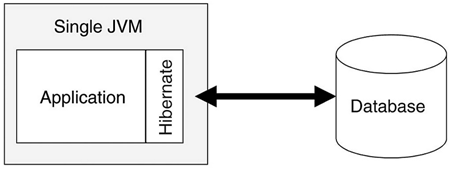
# First-level cache with session object:

1)This is associated with session object. It is the default cache where programmer need not write any extra code.

2)For example if we have to execute multiple statement then no need to hit database again and again but we can use addBatch() method on session object.when the transaction will commit or flushed then hibernates executes all the query with executeBatch() statement.it will increase the performance.

3)Suppose we are using get() method on session object to retrieve the record then it will store in session cache and suppose second time it need the same record then it doesn’t hit the database but it will get from session cache.

When we fetch the same record from different session object then it will hit the database.



# Second-level cache with SessionFactory Object

1. It is associated with SessionFactory object.

2) While running the transactions, whatever records will be fetched through session object that will be stored in SessionFactory cache (buffer).

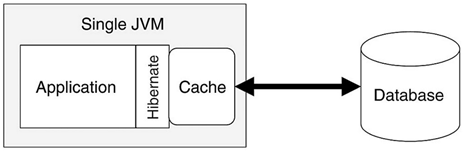
3) So for every session object that is created for SessionFactory, records will be available. Hence whenever required the same records then no need hit the database but it will get from the SessionFactory object.

4) The entire applications can access the **SessionFactory** cache.

5) It is like **ServletContext** in case of servlets.

6) Hibernate comes with four open-source cache implementations to support second-level caching.

1**.** EHCache (Easy Hibernate Cache)  
2. OSCache (Open Symphony Cache)  
3. Swarm Cache  
4. JBoss Tree Cache.



7) For this program of Cache Hibernate, we use **EHCache**. EHCache comes by default with Hibernate software.

8)In second-level cache, we use two different session objects and still database hit is made only once. It requires minor changes in both XML files. One more requirement is a addition of a new XML file **ehcache.xml** .

9) Evicts all second level cache hibernate entites using evictEntity() method.

# Query-level Cache:

**hibernate.cache.use\_query\_cache="true"**

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

<http://way2java.com/hibernate/hibernate-first-level-and-second-level-cache-examples/>