HIBERNATE-NOTES

Frame work:

* A framework is a body or platform of pre-written code which can used by the programmer for the development of an application
* A framework will not do anything by itself and it has to be controlled by a programmer
* A java framework is a collection of classes, interface and some tools

Ex: hibernate, Spring, ibatis, Struts etc.

Types of frame-work:

* Invasive frame-work
* Non-invasive frame-work

Invasive Frame-work:

* A frame-work which will force the programmer to extends its classes and implements its interface is called as invasive frame-work

Ex: EJB, Structs etc.

Non-invasive frame-work:

* A frame-work which will not force the programmer to extends its classes and implements its interface is called as non-invasive frame-work

Ex: hibernate, Spring etc.

Dis-Advantage of JDBC:

* Duplicate code or boiler plate code
* JDBC does not support table creation
* JDBC does not provide any strategy for the generation of primary key
* Fetching the data from the multiple tables by using JDBC requires complex join queries
* JDBC does not support cache mechanism because of which traffic between java application and the data base server increase and efficient decrease

Cache Mechanism:

* It is a temporary layer of storage which is used to store the data to server the feature request

POJO (plain old java object)

Definition:

* Pojo is a class which will not have any special restriction other than those which use imposed by java

Rules are Like:

* Rules of Keyword
* Rules of identifiers
* Rules for constructor

Specification of POJO:

* A POJO class must have public
* I should have a public no argument constructor
* If should have public getter and setter method for all the fields
* If should don’t have extends other class
* If should don’t implement any pre-specified interface

Easy Loading:

* If you use POJO class easy loading will be there since it extends object class hence object class method has to be loaded

Advantage of POJO:

* Easy to understand
* Simple design
* Easy maintained
* Readability can be improved
* We can develop light weight application
* The application which uses POJO implements is known as light weight application

Light-weight Frame work:

* A frame work which make use of POJO implementation is called as light weight application

Example to write POJO class:

public class Employee {

private int id;

private String name;

private String email;

private String password;

private double Salary;

public double getSalary () {

return Salary;

}

public void setSalary (double salary) {

Salary = salary;

}

public int getId () {

return id;

}

public void setId (int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName (String name) {

this.name = name;

}

public String getEmail () {

return email;

}

public void setEmail (String email) {

this. email= email;

}

public String getPassword () {

return password;

}

public void setPassword (String password) {

this. password= password;

}

}

ORM (object relation mapping):

Definition:

* The process of converting an object into relation module (table) is called as object relation mapping
* We can achieve object relation mapping with the help of ORM tool
* An ORM tool will help us to simply the interaction between java application and the data base server
* An ORM tool will provide us a platform where we can deal with the object instead of writing SQL statements
* ORM tool use some configuration file to convert class into table, fields into column and object into records

Entity class (POJO class):

* The class which represents table in the data base server called as entity class

Rules to create entity class:

* An entity class must be public
* An entity class must have no arguments
* An entity class must have getter and setter method for all the field
* An entity class must have a field which represent a primary key

Specification for ORM tool:

* Each entity class represent a table in the data base server
* All fields of entity class represent a column in the table
* All objects of entity class represent a record in the table

Diagram:

Java Application HIBERNATE DATA BASE

ORM TOOL

**Public class Student**

**{**

**Private int id:**

**Private String name:**

**//getter and setter method**

**P S V M (String [] args)**

**{**

**Student s=new Student ();**

**s. setID (101);**

**s. setName (“krish”);**

**save (s);**

**}**

**}**

Mapping

Student

Id name

101 krish

Configuration

Hibernate:

* Hibernate is an open source, non-invasive, light weight ORM tool which is used to achieve object relation mapping
* Hibernate provide a platform which we are directly deal with object instead of writing SQL query
* By using hibernate we can simplify the interaction between the application and the data base server
* Hibernate class implementation JPA, so we can also use JPA specification with the help of hibernate
* CFG file is give the information about the driver class, URL, USER NAME, PASSWORD

Hibernate Architecture:

**JPA**

**OBJECT**

Java Application

DATA BASE

HIBERNATE

**JDBC**

Advantage of hibernate:

* It is an open source frame work
* It is a light weight frame work because it will implement POJO class
* Hibernate support auto table creation
* Hibernate support two level of catch mechanism 1st level and 2 level because of which the traffic between java application and data base server will decrease and efficient will increase
* It supports dialect

Dialect:

* It is a component of ORM tool
* The ORM tool has to communicate with difference version of data base
* It’s like version
* To solve the problem, we have different version of dialect

Example:

MYSQL 5

MYSQL 5.5

HQL (Hibernate Query Language);

* HQL query language is independent data base
* It is a query language which is similar to SQL using which we can write data base in-dependent queries
* To HQL we can use entity class name instead of using the table name in the query

|  |  |
| --- | --- |
| SQL | HQL |
| Select \* from employee | Select e from employee e |
| Select \* from emp where id =? | Select e from employee e where e.id=:id |
| Select \* from emp where salary=? | Select e from employee e where e. salary =?1 |
| Select name from employee | Select e. name from employee e |

Hibernate configuration file:

* If is the file which is used to specify the resource required for hibernate to connect with data base
* This is the file must be created in the src/main/resource and it must be saved with an extension of cfg.xml
* The root tag of this file is <hibernate-configuration>
* The child tag of hibernate configuration is <session-factory>
* <property> is the child element of <session-factory> which is used to configure the properties like driver class, URL, username, password, hbm2ddl.auto, dialect, show\_sql, format\_ sql etc.

Xml. File:

<?**xml** version=*"1.0"* encoding=*"UTF-8"*?>

<!**DOCTYPE** hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"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://localhost:3306/emp?createDatabaseIfNotExist=true</**property**>

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

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

<**property** name=*"dialect"*>org.hibernate.dialect.MySQL55Dialect</**property**>

<**property** name=*"hbm2ddl.auto"*>update</**property**>

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

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

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

</**session-factory**>

</**hibernate-configuration**>

Org.hibernate.cfg.Configuration:

* It is a class belongs to hibernate which represent the hibernate configuration
* The class will provide the method to load the hibernate configuration file and also build the session factory

Following we the important methods of configuration class:

Configure ();

* This method will consider hibernate. Cfg .xml as the default configuration resource and return configuration object
* This method used to load configuration setting from hibernate
* The return type of the method is configuration

Configure (String file):

* The method will consider the argument as the configuration resource return the configuration object

Org. hibernates. Session factory:

* It is an interface belongs to hibernate frame work
* It is used to check the configuration file
* Session factory provide a pool of session
* When your application needs to interact with the data base it typically requests session from session factory

Buildsessionfactory ();

* It is a non-static factory or helper method it is used to execute and return the implementation class object of session factory
* Hence the return type of this method is session factory

To test the configuration:

public class Test-Configuration

{

public static void main (String [] args)

{

Configuration con= new Configuration ();

con. configure ();

SessionFactory sf=con. buildSessionFactory ();

System.***out***.println (sf);

}

}

It will print the address of the object configuration is correct

It is null then configuration has some problem

* Hibernate mapping file:
* The process of mapping an entity class with a data base table is called hibernate mapping
* If you want to map the class with table in the data base, we need to hibernate mapping

Hibernate mapping can be done in two ways

* By using hibernate mapping
* JPA annotation

Hibernate mapping:

* A file which is used to map an entity class with a table in the data base present is called to hibernate mapping file.
* A hibernate mapping file should be saved with extension of.hbm.xml
* It can be created by src/main/resource
* The root tag of hibernate mapping is <hibernate-mapping>
* The <class> tag is the child tag of hibernate mapping which is used to map entity class with the table base
* The <id> is the child tag of class tag which is used to specify the details of primary key it is used to expression primary key column in the table
* <generator> tag is the child tag <id> which is used to provide the generation staggery for primary key
* The <property> tag is the child tag of <class> which is used to specify the details of data base

Following use the important generator classes:

Assigned Sequence Native <sequence Hilo>

Increment Hilo identity

Mapping.hbm.xml:

<?**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=*"org.jsp.HibinsertRecord.Employee"* table=*"employees"*>

<**id** name=*"id"* column=*"sid"*>

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

</**id**>

<**property** name=*"name"* column=*"sname"*/>

<**property** name=*"email"* column=*"email"*/>

<**property** name=*"password"* column =*" password"*></**property**>

<**property** name=*"Salary"* column=*"sal"*/>

</**class**>

</**hibernate-mapping**>

Org.hibernate.session:

* It is an interface present in hibernate frame work
* Session in hibernate like a conversion/interaction between java application to data base
* Session interface provide some method using which we can save, update, delete, fetch the record from the data base

Following are the important method present in session factory:

* Save (object)
* Update (object)
* Saveorupdate (object)
* Get (class<T>, serializable)
* Delete (object)
* Load (class<T>, serializable)

Opensession ();

* It is a method present in session factory interface
* Opensession () create and return the implementation class object of session interface
* Hence the return type of opensession () is session interface

Save ();

* It is a method present in session interface
* This method is used to save the record in the data base
* The method will save the record either by assigning the generator identity or by using the assigned identifier by default and then return the generator identifier
* The return type of the method is serializable
* Save () return the identifier primary key of the stored entity of a serializable object

Org.hibernate. Transaction ();

* It is an interface present in hibernate frame work
* It is used to represent the hibernate transaction

Transaction means: sequence of operation which are transited a single unit of work either all the operation has to be executed successfully or any operation fails than roll back the completed operation from the beginning

Syntax:

Transaction T = session. beignTransaction ();

Transaction T = session. get Transaction ();

Begin Transaction ();

* It is a method present in session interface
* It is a non-static or helper method which is used to create and return the implementation class object of transaction interface
* Hence the return type of begin transaction is transaction interface

Get transaction ();

* Transaction interface provide some methods for transaction in hibernate

Begin ();

Commit ();

Example to understand insert the record in the data base:

public class Insert Record

{

public static void main (String [] args) {

Configuration con= new Configuration ();

con. Configure ();

SessionFactory sf=con. buildSessionFactory ();

System.***out***.println(sf);

Session s=sf.openSession ();

Transaction trans= s.beginTransaction ();

Employee e= new Employee ();

e. set Name("Gowtham");

e. set Email ("@123");

e. set Password ("12345abc");

e. set Salary (4000);

s. save (e);

}

}

Get (class<T>, serializable);

* It is a method present in session factory
* This method is used to fetch a record from the table by using an identifier
* This method will search for the specified identifier in a table which is mapped with the specified entity
* If the table have the specified primary key this method will create an object of 1st argument type it will initialize that object and return the reference else it will return null
* It will throw type mismatch exception if the second argument type is not same as type of primary key in the specified entity class
* It will throw unknown entity exception if the argument is not an entity it will throw illegal argument exception if the second argument is null
* The return type of the method is same as the type of 1st argument
* Ex: if we pass employee. Class as an argument then return type would be employee

Example to understand get method:

public class GEt

{

public static void main(String[] args) {

Configuration con= new Configuration ();

con. configure ();

SessionFactory sf=con. buildSessionFactory ();

Session s=sf. openSession ();

Employee e=s. get (Employee.class, 1);

If (e! =null)

{

System.***out***.println(e.getName ());

System.***out***.println(e.getEmail ());

System.***out***.println(e.getPassword ());

}

else

{

System.***out***.println("INVALID ID NUMBER");

}

}

}

Load (Class<T>, serializable);

* It is a method present in session interface
* This method is used to fetch a record from the table given by using identifier (primary key)
* This method is assuming the identifier is present and return the proxy object by assign the passed identifier
* This method will initialize the object only when we try is used the object which is not there in data base
* It will throw unknown entity exception if the argument is not an entity it will throw illegal argument exception if the second argument is null
* The return type of the method is same as the type of 1st argument
* Ex: if we pass employee. Class as an argument then return type would be employee

Example to understand load method:

public class Load

{

public static void main(String[] args) {

Configuration con= new Configuration();

con.configure();

SessionFactory sf=con.buildSessionFactory();

Session s=sf.openSession();

Employee e=s.load(Employee.class, 1);

if(e!=null)

{

System.***out***.println(e.getName());

System.***out***.println(e.getEmail());

System.***out***.println(e.getPassword());

}

else

{

System.***out***.println("INVALID ID NUMBER");

}

}

}

Difference between get and load method?

|  |  |
| --- | --- |
| Load | Get |
| Load method with return a proxy object by assigning the passed identifier | It will return an initialized object if the identifier is present else it will return null |
| Load is Lazy | Get is Eager |
| Can not be used to check the existence of record | Can be used the existence record |
| Is faster than get () | Get () is slower than load () |

SaveorUpdate (Object);

* This method is present in session interface
* This will either than save or update a record in the data base
* This will save the record if the identifier is not present and it will update the record of the identifier is present
* Hence if the generator class is assigned and if the identifier not present the method will save () in the data base server

Note:

* Both update and save or update work same if the identifier is present
* The use saveorupdate ()

Generator class must be assigned

It is identified it will not work check once the change into assigned

Example to understand saveorUpdate (Object);

public class Update

{

public static void main(String[] args)

{

Configuration con = new Configuration();

con.configure();

SessionFactory sf= con.buildSessionFactory();

Session s=sf.openSession();

Transaction ts= s.beginTransaction();

Employee e= new Employee();

e.setId(1);

e.setName("kishore");

e.setEmail("@12345");

e.setPassword("123467");

s.saveOrUpdate(e);

ts.commit();

}

}

Destroyed ();

* This method is declared in session interface
* This method is used to delete a record from the data base server
* From you fetch the object from the data base server using get () based you delete
* Hence if you want to delete the record we used to pass the configure object
* It will throw illegal argument exception of the argument is not an object if that required is null

Note:

It not possible because that it is not responding any record in the table you will get illegal argument exception with root cast mapping exception

Example to understand delete ();

public class Delete

{

public static void main(String[] args)

{

Configuration configuration= new Configuration ();

configuration.configure();

SessionFactory sf= configuration.buildSessionFactory();

Session s= sf.openSession();

Transaction ts=s.beginTransaction();

Employee e= s.get(Employee.class, 3);

s.delete(e);

ts.commit();

}

}

All curd operation completed:

* The important with request to get () and load () are both methods use primary key to fetch record
* If I ask you fetch the record based on the name, password, phone, email hence it is impossible

Because none of them primary key:

* If you want fetch all the record from the employee table these is no in-build method
* Then the solution is SQL it is similar to SQL but in SQL we write table name but in HQL we write entity class name
* SQL is data base dependent where as HQL is data base in-dependent

Org.hibernate. Query Query<T>;

* It is an interface present in hibernate frame work
* This interface is used to execute the HQL query
* We can fetch the result from the data base server with the help of method which are present in the interface which we can use HQL statement or query
* The implementation object of the interface can be created with the help of create query (String) which is present in session interface

Syntax:

Query<T> q= session. createQuery (String HQL);

Note:

Hence the type of query it depends on the result

* If the query is returning employee object, then it is employee
* If it is returning integer then it is integer

Following are the important method present in query interface:

getResultList ();

* This method will return list of record by query object
* We should call the method in the query is returning more than one result the return type of the is java. Util. list
* The size of the list will be same as the number of returns by the Query
* If there is no result of the list it size will be zero

Example to understand getResultList ();

public class ResultList

{

public static void main(String[] args)

{

Configuration con = new Configuration();

con.configure();

SessionFactory sf = con.buildSessionFactory();

Session s= sf.openSession();

Query<Employee> q= s.createQuery("select e from Employee e");

List<Employee> l1=q.getResultList();

For (Employee e:l1)

{

// System.out.println(e);

System.***out***.print(e.getId()+" ");

System.***out***.print(e.getName()+" ");

System.***out***.print(e.getEmail()+" ");

System.***out***.print(e.getPassword());

System.***out***.println();

}

}

getSingleResult ();

* we should call the method if the query return exactly one result
* This method will return a single result returned by Query
* It will throw No Result exception if the Query doesn’t have and result
* If throw no unique Result exception if the Query have more than one result
* The return type of the method is same as the type of the Query

Example to understand get single result method:

public class BasedColumn

{

*@SuppressWarnings*({ "unchecked" })

public static void main(String[] args)

{

Configuration con = new Configuration();

con.configure();

SessionFactory sf = con.buildSessionFactory();

Session s= sf.openSession();

Query<String> q1=s.createQuery("select e.name from Employee e where e.id=:id" );

q1.setParameter("id", 6);

try {

String s1 = q1.getSingleResult();

System.***out***.println(s1);

} catch (NoResultException e) {

System.***out***.println("invalid");

}

}

Hibernate query parameter:

* These are used to hold the dynamic value from the user during the execution time in an HQL statement

We have two types of parameters:

* Named parameter (name)
* Number parameter (integer)

We must assign the value for the hibernate parameter:

The set value we have following in query interface:

Set parameter (int position, object value);

It is used to assign the value for named parameter

Query<String> q2=s.createQuery("select e.name from Employee e where e.id=?0 and email=?1" );

q2.setParameter(0, 1);

q2.setParameter(1, "@12345");

Set parameter (String name, object value);

It is used to assign the value for a named parameter

Query<String> q1=s.createQuery("select e.name from Employee e where e.id=:id" );

q1.setParameter("id", 6);

JPA ANNOTATION

Java persistence Api:

Till now we have to do mapping using mapping file in hbm.xml. file into map on entity class with which in the data base

But hibernate mapping can be in two different ways;

* By using hibernate mapping file
* By using JPA annotation

Previous we were using hibernate mapping file but we need to use annotation

Note:

* If you map the entity class with table in the data base using hibernate mapping hibernate.xml to hibernate configuration file we need to use mapping resource
* If you map entity class with the table in the data base server by using JPA annotation in hibernate configuration file we need to use mapping class
* There is the annotation which are used to map an entity class with a table in the data base server these annotations are present in javax. persistence package

Following is the important annotation which are used to map an entity class with a table in the data base server:

@Entity:

* It is a class level annotation belongs to JPA and present in javax. Persistence package
* This annotation is used to mark the class an entity class. This entity class in going to map as table in the data base server because of ORM
* This class which is annotated with @entity will represent a table in the data base

@table:

* It is a class level annotation belongs to JPA and present in javax. Persistence package
* Since we are using @ entity annotation the annotated class in going to represent a table
* To provide the details of the table we use this

@ID:

* It is a class level annotated belong to JPA and present in javax. Persistence package
* It is used to mark the field to a primary key

@GeneratedValue:

* It is an annotation belongs to JPA and present in javax. Persistence package
* This annotation is used to mark the field as generator value we can provide the strategy for the generation of primary key by using the Enum javax. Persistence generation type which belongs to JPA

Following is the important generation type:

* Generation type Identity
* Generation type Sequence
* Generation type Auto
* Generation type table

@column:

* It is an annotation belongs to JPA and present in javax. Persistence package
* We use this annotation just above the field of entity class to provide the information about the column which is mapped with the annotation field

Following is the important attribute of a column:

Name Nullable Precision Unique

Length

Steps to map User class with user table by JPA annotation:

Step1: create a maven project

Step2: Add the maven dependency

* + Hibernate core relocation
  + MYSQL connector java

Step3: Create an entity class by name user

Step4: Use JPA annotations

Step5: Create hibernate configuration file (hibernate.cfg.xml)

Step6: Create Test Configuration

Only problem with hibernate is Tight coupling

We have used Configuration class

Session Factory

Session

Transaction

Query interface with respect to hibernate

**Then the solution is JPA**

* It is the specification does not have any implementation
* Using JPA we can achieve loose coupling which can be implemented by orm tools

**We can use:**

* JPA with Hibernate
* JPA with struts
* JPA with top link

Find (class<T>, object primary key)

* It is a method present in entity manager interface
* This method is used to fetch a record from the table by using primary key
* This method will search for the specified primary key in the table which is mapped with the specified entity class
* If the primary key is present, it will return the reference of the object else it will return null
* The return type of this method is same as the type of the first argument
* If it is Employee-->it will be Employee

Note: if you wrongly enter persistence unit name you will get --->Persistence exception

* It will throw illegleArgumentException with the root cause unknownEntityException if the first argument is not an entity
* It will throw illegleArgumentException with the root cause TypeMisMatchException if the second argument type is different from the type of primary key in the entity class
* It will throw illegleArgumentException if the second argument is null

Example:

public class Fetch\_BasedonId

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

System.***out***.println(emf);

Entity Manager em=emf. createEntityManager ();

User u=em. Find (User.class, 1);

If (u! =null) -->**Avoid null pointer exception**

{

System.***out***.println(u. get Id ());

System.***out***.println(u. get Email ());

System.***out***.println(u. get Name ());

System.***out***.println(u. getPhno ());

}

else

{

System.***out***.println("invalid id value");

}

}

}

Update a record in the database:

There are 2 different ways in which we can update a record in the data base

* Fetch (using find ()) and update
* Use merge () of entity manager interface

Best way is fetch and update

* Any modification on the persistence object will directly affect the record in the table

Merge ():

* It will consider the generated id while updating if the id is present then it will update else it will save the record

Code to update a user by using fetch and update way:

public class Update User

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

System.***out***.println(emf);

Entity Manager em=emf. createEntityManager ();

Entity Transaction trans=em. get Transaction ();

trans. Begin ();

User u= em. Find (User.class, 1);

If (u! = null)

{

u. set Name ("Krish");

}

else

{

System.***err***.println("invalid id value ");

}

trans. Commit ();

}

}

Merge (T entity)

* It is a method present in entity manager interface
* This method is used to merge the state of the passed object with the record present in the data base server
* This method is will save the record by assigning the generated id or by using the assigned id if the primary key is not present
* It will update the record if the primary key is present
* The return type of this method is same as the type of the argument

**Code to update a record in the merchant table using merge (object)**

Update merchant by merge. Java class

* Create an object of merchant class
* Enter the id value (m. setId (1) or m. setId (sc. Next Int ())
* Use merge (m) of entity manager interface
* Don’t use @generated value // to save the assigned value
* If the given id is not present and if the generation strategy is identity, then it will not throw any exception but the record will be saved with identity strategy, assigned id will not be saved

**Code for merge ();**

public class Update User

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

EntityManager em =emf. createEntityManager ();

EntityTransaction trans=em. Get Transaction ();

trans. Begin ();

User u= new User (); //---> Transient state

u. set Name("rahul");

u. set Id (10);

u. set Email("axy@gmail.com");

u. set Password ("123");

u. set Phno (6380609823l);

em. merge(u);

trans. Commit ();

}

}

Remove (object)

* This method is present in entity manager interface
* This method is used to remove a persistence object (record) from the data base server
* If we pass a detached or transient object to this method it will throw illegal Argument Exception

**Code to delete a record from merchant table by using remove (object):**

Find the merchant by using find (merchant.class,1)

If (m! = null)

{

Manager. remove (m);

Em. Commit ();

]

Example:

public class Delete user

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

System.***out***.println(emf);

EntityManager em=emf. CreateEntityManager ();

EntityTransaction trans=em. getTransaction ();

trans. Begin ();

User u=em. Find(User.class, 3);

If (u! = null)

{

em. Remove (u);

trans. commit ();

}

else

{

System.***out***.println("the record is not present in the table so can’t delete");

}

}

}

**Notes:**

**As there is no build in method are available in entity manager to fetch a record without using id**

**Now we need to go for Query interface**

* **In JPA Query interface is present in javax. Persistence package**
* **In hibernate Query interface is present in org. hibernate. Query package**
* **JPA uses JPQL**
* **Hibernate uses HQl**
* **In hibernate Query interface accept Generic (Query <T>)**
* **In JPA Query interface does not accept any generic (Query)**
* **The initial version of JPA were designed before the Java introduced generic in JDK 1.5 to maintain compatibility with older version of java**
* **To support the developer who were using the older version of Java JPA Query interface was designed without generic**

**Javax. Persistence. Query:**

**It is an interface belongs to JPA which is used to execute JPQL query**

**The implementation object of this interface can be created by using following methods which are present in entity manager**

1. **create Query (string)**
2. **createNamedQuery (String)**
3. **createNativeQuery (String)**

**Syntax:**

1. **query q = em. createQuery (JPQL Query)**
2. **Query q= em. createNamedQuery (String name of the JPQL Query)**
3. **Query q = em. createNativeQuery (String SQL Query)**

**Following are the important methods present in query interface:**

**Object getSingleResult ();**

* **If the query returning single result, then we need to use this method to fetch that single result**
* **It throws NoresultException if the query is not returning any result**
* **It throws nouniqueresultException if the query is returning more than one result**
* **The return type of this method is java. lang. Object**
* public class JPQL\_Tofetch
* {
* public static void main (String [] args)
* {
* EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");
* EntityManager em=emf. createEntityManager ();
* Query q= em. createQuery ("select u from User u where u. phno =?1 and u. password=?2");
* q. setParameter (1, 91768969403l);
* q. setParameter (2, "123abc");
* try
* {
* User u1=(User) q. getSingleResult ();
* System.***out***.println(u1.getName());
* }
* catch (NoResultException e)
* {
* System.***out***.println("NO Result Found for that phno and password");
* }
* }
* }

**List <T> getResultList ();**

* **If the query returning the list of result, then to fetch that list of result, we need to use the method**
* **If the query returning no list, it will not throw any exception it will return empty list**
* **The return type of this method is java. util. List**

public class JPQL\_Tofetch

{

public static void main(String[] args)

{

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("dev");

EntityManager em=emf. createEntityManager ();

Query q= em. createQuery ("select u from User u ");

List<User> l= q. getResultList();

If (l. size ()>0)

{

for (User user: l)

{

System.***out***.println(user. get Name ());

}

}

else

{

System.***out***.println("No user Found");

}

}

}

**set Parameter (int, object):**

**it is used to assign the value for a numbered parameter**

* Query q= em. createQuery ("select u from User u where u. phno =?1 and u. password=?2");
* q. setParameter (1, 91768969403l);
* q. setParameter (2, "123abc");

**set Parameter (String, object):**

**it is used to assign the value for a named parameter**

* Query q= em. createQuery ("select u from User u where u. phno =:ph and u. password=:ps");
* q. setParameter (ph, 91768969403l);
* q. setParameter (ps, "123abc");

**Create Query (String JPQL)**

* **It uses JPQL query as a parameter**
* **If we want to give the name of the query instead of writing the query directly then use the below is**

**CreateNamedQuery ();**

* **Create named query (String name of the JPQL Query);**
* **It uses name of the JPQL query as a parameter**
* **To use the above method, you need to use one of the annotations in the entity class just above the @entity annotation**

**Example:**

*@NamedQuery* (name = "details", query = "select u from User u")

**@ NamedQuery:**

* **It is a class level annotation belongs to JPA present in javax. Persistence package**
* **It is used to create a Named JPQL Query For an entity class**

**Following is the important attribute of this annotation**

* **Name**
* **Query**

**@NamedQuerys:**

* **It is n class level annotation belongs to JPA and present in javax. persistence package**
* **It is a container annotation that is used to aggregate/group multiple @NamedQuery annotation**
* **Value attribute of this annotation is used to store multiple @NamedQuery annotation**
* **Value attribute represent ----> array of @NamedQuery annotation**

**Example:**

*@NamedQueries*({

*@NamedQuery* (name = "details", query = "select u from User u"),

*@NamedQuery* (name="name", query = "select u.name from User u where u.name=?1"),

*@NamedQuery* (name = "findname", query = "select u.phno from User u")

})

**Example program to understand create named query:**

*@Entity*

*@NamedQueries* ({

*@NamedQuery* (name = "details", query = "select u from User u"),

*@NamedQuery* (name="phoneno", query = "select u.name from User u where u.name=?1"),

*@NamedQuery* (name = "findname", query = "select u.phno from User u")

})

public class User

{

//provide fields

// getters and setters’ method

}

public class NamedQuery

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

EntityManager em=emf.createEntityManager ();

Query q=em.createNamedQuery ("details");

List<User> result = q. getResultList ();

If (! result. isEmpty ())

{

For (User u: result)

{

System.***out***.println(u. getName ());

}

}

else

{

System.***out***.println(result);

}

**CreateNativeQuery:**

* This method uses SQL query to execute
* Hence, we don’t pass JPQL query
* It is data base dependent

Example program to understand create Native Query:

* Here we use SQL not HQL/JPQL
* Find all User. Java
* In this code if we use createNativeQuery (“select \* from User”) only, we will get class cast Exception --> so we pass the result type also

CreateativeQuery (“select \* from User”, User. Class);

public class NativeQuery

{

public static void main (String [] args)

{

EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

EntityManager em=emf. createEntityManager ();

Query q= em. createNativeQuery ("select \* from User", User.class);

/\* if you want to use sql query to use craetive Native method

\* If the query returning only one coloum the return type is that data type \* more than one the it is object []

\*

\*/

List<User> u= q. getResultList ();

If (! u.isEmpty())

{

For (User u1: u)

System.***out***.println(u1);

}

else

{

System.***out***.println("invalid ");

}

}

Association mapping:

* In case of SQL, we were having emp table and dept table
* Dept no is used to develop (build) relationship between tables
* Till now we have not built the relationship between the tables in hibernate/JPA
* The process of establishing the relationship between the data base table is called as association mapping
* Association mapping refers to how relationship between entities (Java objects) are mapped to data base tables

The association mapping can be broadly classified into two types:

* Uni-Directional Mapping
* Bi-Directional Mapping

Uni-Directional mapping:

* In uni-directional mapping we can access the child entity with the help of parent entity and vice versa is not possible (Only one entity is aware of relationship)

Ex: One person (parent entity) can have on Aadhar card (child entity)

Bi-directional mapping:

* In bi-directional mapping we can access the child entity with the help of parent entity, the parent entity with the help of child entity
* We can further classify the association mapping into following types based on the number of entities associated with each entity

1. One to one
2. One to many
3. Many to one
4. Many to many

One to One:

* In one-to-one association mapping one instance (object) of an entity will be associated with one instance of another entity class

Ex: one person can have only one pan card

One to many:

* One instance of an entity class will be mapped with many instances of another entity class

Ex: Department – employees

One department have many employees

Many to one:

* Many instances of an entity class will be associated with one instance of another entity class

Ex: many answers belong to one Question

Many to many:

* Many instances of an entity class will be mapped with many instances of another entity class

Ex: student-batch

A student can attend many Batches, each batch can have many students

One to one uni-Directional mapping:

Person class pan card class

Id (this is person id) id (this is pan card id)

Name number (string)

Phone (long) dob (local date)

Pincode (int)

* There is no relationship here
* So, we need to build the relationship
* Between the person and pan card we need to build the relation i.e. one-to-one uni
* We need to declare a field in the person class which is of pan card type
* We need to annotate this field with @ oneToone in person class

Example:

*@Entity*

public class Person

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private String email;

private long phno;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public long getPhno() {

return phno;

}

public void setPhno(long phno) {

this.phno = phno;

}

}

*@Entity*

public class Pancard

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String number;

private int pincode;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNumber() {

return number;

}

public void setNumber(String number) {

this.number = number;

}

public int getPincode() {

return pincode;

}

public void setPincode(int pincode) {

this.pincode = pincode;

}

}

Because of these 2 tables are going to create

Person table

Id (pk | name | phone | card\_id (fk)) ----> these references pan card

Pan card table

Id (pk | number | dob | pin code)

@oneToone

* It is an annotation belongs to JPA and present in javax. persistence package
* This annotation is used to map one instance of an entity class with one instance of another class entity
* We will get exception, if we use this annotation in collection

Example program to understand one-to-one uni-directional mapping:

*@Entity*

public class Person

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private String email;

private long phno;

*@OneToOne* (cascade = *CascadeType*.***ALL***)

private Pancard card;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public long getPhno() {

return phno;

}

public void setPhno(long phno) {

this.phno = phno;

}

}

Cascade:

It is an attribute used in association mapping which we can modify the state of child entity whenever the state of parent entity is modified

This attribute is present in the following annotations:

1. @One to one
2. @One to many
3. @Many to one
4. @Many to many

We can assign the value for cascade attribute by using the Enum javax. Persistence. cascade Type which is present in JPA

Following are the different cascade types available:

* Cascade Type. persist (To save);

The child entity will get saved along with the parent entity if we use this cascade type

* Cascade Type. merge(To update);

This child entity will get update along with the parent entity if we use this cascade type

* Cascade Type. Move (To Delete);

This child entity will get delete along with the parent entity if we use this cascade type

* Cascade Type. Detach (To Disconnect);

This child entity will get disconnect from the session (entity manager) if we use this cascade type

* Cascade Type. Refresh (To Refresh);

This child entity will get refreshed along with the parent entity if we use this cascade type

* Cascade Type. ALL (Complete);

Cascade type ALL ensure that all operation (persist, merge, remove, refresh, detach) on an entity are cascade to its associated entity

Example to understand cascade type:

*@Entity*

public class Person

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private String email;

*@OneToOne* (cascade = *CascadeType*.***ALL***)

// @JoinColumn(name = "card\_id",referencedColumnName = "id")

private Pancard card;

one to one Bi-directional mapping:

In this example we will understand the concept of Owning side (Join column) and non-owning side (Mapped by)

Example:

User and Aadhar card

User Aadhar Card

-id -id

-name -number

-phone -dob (local date) card.set (local date. parse(“1993-02-17”)

@one to one -address

Private Aadhar card card @One to One

Private User user:

We specify owning side of Association mapping with --------@joincolumn annotation

@join column is an annotation used in hibernate to specify the column that will be used to join two tables in the data base

* It is used to represent the column in the data base that acts as the foreign key
* Owning side and non-owning side is only applicable for bi-directional mapping

Notes:

There is no concept of mapped by attribute in many to one association mapping

There is no rule to decide owning and non-owning it totally depends on the requirement

@join column:

* It is an annotation belongs to JPA and present in javax. Persistence package
* We choose one entity to manage the relationship we call this owning side
* It is used to provide the details of foreign key in association mapping. It can also be used to specify the owning side of association in bi-directional mapping
* It must used with the annotation which are used to achieve association mapping

Mapped By:

* It is an attribute used in annotation mapping to specify the non-owning side of association in bi-directional mapping
* It indicates that the field is associated with is not the owner of the relation ship is managed by the other entity. This helps hibernate understand which entity is responsible for managing the relationship

Mapped/attribute is there in

@OneToOne

@OneToMany

@ManyToMany

Example to understand bi-directional:

*@Entity*

public class User

{

*@Id*

*@GeneratedValue* (strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private long phno;

*@OneToOne*(cascade = *CascadeType*.***ALL***)

*@JoinColumn*

private AadharCard card;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public long getPhno() {

return phno;

}

public void setPhno(long phno) {

this.phno = phno;

}

public AadharCard getCard() {

return card;

}

public void setCard(AadharCard card) {

this.card = card;

}

*@Override*

public String toString() {

return "User [id=" + id + ", name=" + name + ", phno=" + phno + "]";

}

}

*@Entity*

public class AadharCard

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private long number;

private String Address;

*@OneToOne*(mappedBy = "card")

private User user;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public long getNumber() {

return number;

}

public void setNumber(long number) {

this.number = number;

}

public String getAddress() {

return Address;

}

public void setAddress(String address) {

Address = address;

}

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

*@Override*

public String toString() {

return "AadharCard [id=" + id + ", number=" + number + ", Address=" + Address + "]"}}

public class Save

{

public static void main(String[] args)

{

EntityManagerFactory emf= Persistence.*createEntityManagerFactory*("dev");

EntityManager em=emf.createEntityManager();

EntityTransaction trans =em.getTransaction();

trans.begin();

User u = new User();

u.setName("tara");

u.setPhno(6380654823l);

AadharCard card =new AadharCard();

card.setNumber(6380124567811l);

card.setAddress("thanjavur");

card.setUser(u);

u.setCard(card);

em.persist(u);

trans.commit();

}

}

Imp for example:

@OneToOne (mapped by= “card”)

Private User user;

This tells card field in the User class is responsible to build relationship

The mapped by attribute in hibernate

@OneToMany Uni-Directional:

Example: one department can have many employees

Department. Class Employee. Class

Int id; int id;

String name; String name;

String location; String desg;

@OneToManys

List <Employee> emps;

Points:

* It is an annotation belongs to JPA and it present in javax. Persistence package
* This annotation is used to map one instance of an entity class with many instances of another entity class
* We will get annotation exception if we use this annotation on a field which is not a collection

1. e: the second entity must be collection type

Note:

* Ef Codd rule (one cell must always only one value)
* Primary key cannot be duplicated
* In onetomany uni-directional mapping we get 3 table but the 3rd table contains the primary key of 1st table and 2rd table
* 3rd table is called as join table (Department\_Employee)

Third table columns:

Department\_id

Employee\_id

* We can not avoid the creation of 3rd table in one-to-many uni-directional
* Here 3rd table is used to established a relationship between other two tables

Note:

In case of OneToOne ---------------------> we will get extra column to build relation

ManyToOne

In case of OneToMany ----------------------> we will get extra table

ManyToMany

Many To One Uni-Directional mapping:

Example:

Many answers belong to one question

Questiondata.class answerdata.class

Int id; int id;

String question; String answer;

String posted by; String answer by;

@ManyToOne

Private Questiondata data;

Questiondata answer data

|  |  |  |
| --- | --- | --- |
| id | question | posted by |
|  |  |  |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| id | answer | answer by | Data id |
|  |  |  |  |
|  |  |  |  |

@ManyToOne;

* It is an annotation belongs to JPA and present in javax. Persistence package
* This annotation is used to map many instances of one entity class with one instance of another entity class
* We will get annotation exception if we use it on collection

Example program to understand Many to One uni-directional mapping:

Example program to understand Oneto Many or Many Toone Bi-directional:

*@Entity*

public class Merchant

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private long phno;

private String gst\_number;

private String email;

private String password;

*@OneToMany*()

private List<Product> products;

}

*@Entity*

public class Product

{

*@Id*

*@GeneratedValue*(strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private String Brand;

private String category;

*@ManyToOne*

*@JoinColumn*

private Merchant merchant;

}

Here in this case, we will get 3 tables (whenever we established a relationship)

1, Merchant

Id | name | gst\_number | phone | password

2, product

Id | name | brand | category | merchant\_id (fk) this we will =get because of many to one

3, merchant product

Merchant\_id | productid

Already merchant\_id column is there in the product table

No need of 3rd table

@Many To Many

Ex: Many Batches are attended by Many students

Batch Students

Id; id;

Batch code; name;

Trainer; phone;

Subject; perc;

@ManyToMany (cascade = Cascade. Type. All)

List<Students> stud;

Note:

We cannot avoid the creation of 3rd table since it is required to build relationship between batch and student table

@Many To Many bi-directional:

Batch students

Id; id;

Code; name;

Trainer phone;

Subject; percentage;

@many to Many @Many to many (mapped by student)

@join table

List<students> student; list<batch> batch;

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | Phone | percentage |
|  |  |  |  |
|  |  |  |  |

Batch table student table;

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Code | Trainer | Subject |
|  |  |  |  |
|  |  |  |  |

@ join table:

* It is annotation belongs to JAP and present in javax. Persistence package
* It is used to represent the owning side of association in bi-directional mapping
* It is used to provide the details of join table the table which you want to retain in association mapping

Following is the important attribute present in join table

* Name: it is used to specify the table name
* Join column: it is used to provide the details of the column in the join table which is referring to the owning side of association
* Inversive join column: it is used to provide the details of the column in the join table which is referring to the non-owning side of association

Fetch attribute:

Whenever we load the parent entity whether the child entity has to be loaded or not

Points:

It is an attribute used in association mapping using which we can decide whether the child entity has to be loaded along with the parent entity or not

This attribute present in

@One To One

@One To Many

@Many TO One

@Many To Many

We can assign the value for fetch attribute by using Enum fetch type

Following are the two types of Enum is available:

Fetch type. Eager

If we try to fetch the parent entity and the child entity not be loaded

Fetch type. Lazy

If we try fetch based on parent and child entity that time will be load

Even through we are not using fetch attribute in association mapping some default fetch types are available

Which are as follows

@One To One Eager

@One To Many Lazy

@Many TO One Eager

@Many To Many Lazy

When to use cascade:

Whenever we modify the state of parent entity then the state of child entity has to be modified then we use cascade

Example program to understand fetch attribute using one to one:

Here we have user class and Aadhar card class

In the case of one-to-one relation by default fetch type is eager

So, whenever we try to find the person without any fetch attribute along with the user Aadhar card will be loaded in the console like below

*@Entity*

public class User

{

*@Id*

*@GeneratedValue* (strategy = *GenerationType*.***IDENTITY***)

private int id;

private String name;

private long phno;

*@OneToOne*(cascade = *CascadeType*.***ALL***,fetch = *FetchType*.***LAZY***)

*@JoinColumn*

private AadharCard card;

//getter and setter method

}

public static void main (String [] args)

{ EntityManagerFactoryemf=Persistence.*createEntityManagerFactory*("dev");

EntityManager em= emf. createEntityManager ();

User u=em. Find (User.class,1);

if (u! =null)

{

System.***out***.println(u);

}

else

{

System.***out***.println("Record Is Not Found");

}

}

Output:

Fetch only the user information;

Hibernate:

select

user0\_.id as id1\_6\_0\_,

user0\_.card\_id as card\_id4\_6\_0\_,

user0\_.name as name2\_6\_0\_,

user0\_.phno as phno3\_6\_0\_

from

User user0\_

where

user0\_.id=?

Time stamp:

* Timestamp is a data type used to represent (store) date and time.
* It is commonly used to track the event or operation which are performed based on data and time

Points:

These are the annotation which are used in hibernate to generate the date and time to assign the value for annotated field

These annotations are present in org. hibernate. Annotation package

Following is the important time stamp annotation:

@creation Time Stamp

* This annotation will assign the annotated field with in the entity class with the current virtual machine date and time whenever we save the entity for the first time
* The field in the entity class which is annotated with @CreationTimeStamp will hold the current virtual machine date and time whenever we try to persist the entity into database
* This time and date which is inserted into the database server will not be updated or modified afterwards every through if you update the other field of entity

@Update Time Stamp

* This annotation will assign the annotated field with the current virtual machine date and time when we save or update the entity
* i.e. this field which is annotated with @UpdateTimeStamp in the entity class can be updateable later

composite key:

The combination of more than one column to form a primary key is called as composite key

Note:

If we combine more than one column to form a composite key the combination of those column must be unique in every record in the table

Example:

Where one column is not enough to identify the record uniquely in the record

BTM Data Base

|  |  |
| --- | --- |
| Cid | Cname |
| 101 | J2EE |
| 102 | Java |

Student table Course table

|  |  |
| --- | --- |
| sid | Sname |
| 1 | Guru |
| 2 | Raj |

|  |  |  |  |
| --- | --- | --- | --- |
| Sid | Cid | Sname | Cname |
| 1 | 101 | Guru | J2EE |
| 2 | 102 | Raj | Java |
| 1 | 102 | Guru | Java |
| 2 | 101 | Raj | J2EE |

Master Table

If you want to save the data in master table than you should create an entity class (Master) like below

Now our master class is composite key

It is a rule than composite key class must implements serializable but it violets the rule of POJO

@Entity

Class Master implements Serializable

{

@ID

Private int sid;

@Id

cid;

sname;

cname;

It is not recommended to have two primary keys in an entity class for below reason

Normalization:

Having two sperate primary key implies that each one independent identifies a unique record which can violate normalization principle proper normalization usually involves ensuring that a table has a single unique primary key

Referential integrity:

Foreign key constrains and other relational data base feature are designed to work with a single primary key having multiple primary keys would complicate this relationship and potentially lead to data integrity issues

@embeddable:

* It is a class level annotation belongs to JPA and present in Javax persistence package
* @embeddable annotation is used to indicate the class can be embedded within an entity
* It is used to mark the class as embeddable class
* An embedded class will have all the field which are responsible for the creation of compositive key
* An embedded class must implementation java.io. serializable interface

@embeddedid

* It is an annotation belongs to JPA and present in javax. persistence package
* This annotation is used to mark the field as embedded id (compositive key)
* This field which is annotated with @embeddedble must be of serializable type they we will get exception

Example program to understand @embedale and @embedded id

Hibernate Life Cycle

Hibernate life cycle represent the different states of an object in hibernate

Following is state of an object in hibernate life cycle

Transient State

* An object is said to be transient state when it is newly created
* An object in the transient state will not represent any record in the table modification on the transient object will not have an effect on the record in the table

Persistence State

* An object is said to be persistence state once it is connected with the session
* An object in the persistence state will represent a record in the data base server
* Any modification on the state of persistence object will directly affect the record in the table

Detached State

* An object is said to be in detached state once it is disconnected from the session (entity manager)
* So, modification of detached object will not affect any record in the data base server

Removed State

* An object said to be in removed state once it is deleted from the data base server, we can not delete an object in the detached state by remove(object)
* Remove (object) can only delete persistence object
* Delete method can delete the object which is present in both persistence and detached State

Cache mechanism:

It is a temporary layer of storage which is used to store the data to serve future request

Hibernate support 2 levels of cache mechanism using which we can reduce the traffic between the java application and the data base server and increase the performance

Following is the two level of cache mechanism:

First level cache:

* It is a layer of storage assigned by the hibernate for every session (Entity Manager) to store the data to serve the future request
* Every session will have a dedicated first level cache memory and all of them will be connected with second level cache

Second level cache:

* It is a layer of storage which is shared by all the session (Entity Managers) created by a Session Factory (Entity Managx er Factory)
* By default, only first level cache is supported by hibernate and the 2nd level cache has to be enabled explicitly

Hibernate project:

Design pattern

It is an optimized solution for various technical problems that we may face while creation of the project

Totally there are 3 different types of design pattern:

Creation design pattern:

It is used to write the object creation logic

Ex:

* Singleton design pattern
* Factory design pattern
  + Begin Transaction (). Get transaction ()

Structural design pattern

Behaviour design pattern