Day3: Basic CRUD operation using JPA with Hibernate:

Note: When we call the **createEntityManagerFactory(-)** method on the **Persistence** class by supplying the persistence-unit name, we will get the **EntityManagerFactory** object.

- This method loads the **"persistence.xml"** file into the memory
- **EntityManagerFactory** object should be only one per application.

This **EntityManagerFactory** object contains:

Connection pool (readily available some JDBC connection objects)

Some meta information

This EntityManagerFactory is a heavy-weight object, by using this EntityManagerFactory class only we create the **EntityManager** object.

EntityManagerFactory is a heavy weight object, it should be one per application.

```
EntityManager em= emf.createEntityManager();
```

Note:- Inside every DAO method(for every use case) we need to get the EntityManager object after performing the database operation for that use case we should close the EntityManager object.

Inserting a Record:

In order to perform any **DML (insert update delete)** the method calls should be in a transactional area.

em.getTransaction() method returns the "**javax.persistence.EntityTransaction** " object.

This **EntityTransaction** object is a singleton object, i.e. per EntityManager object, only one EntityTransaction object is created.

To store the object we need to call **persist(-)** method on the **EntityManager** object.

Example:

```
package com.masai;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Demo {

  public static void main(String[] args) {
```

```
    EntityManagerFactory emf= Persistence.createEntityManagerFactory("studentUnit");


    EntityManager em= emf.createEntityManager();

    Student student= new Student(30, "Ratan", 500);


//    EntityTransaction et= em.getTransaction();
//
//    et.begin();
//
//    em.persist(student);
//
//    et.commit();


    em.getTransaction().begin();

    em.persist(student);

    em.getTransaction().commit();

    System.out.println("done...");

    em.close();


  }

}
```

Delete Operation:

```
public class Main {

  public static void main(String[] args) {

    EntityManagerFactory emf=Persistence.createEntityManagerFactory("studentUnit");

    EntityManager em= emf.createEntityManager();


    Scanner sc=new Scanner(System.in);

    System.out.println("Enter roll to delete ");
    int roll=sc.nextInt();

    Student student= em.find(Student.class, roll);

    if(student != null){

      em.getTransaction().begin();

      em.remove(student);

      em.getTransaction().commit();


      System.out.println("Student removed....");

    }else
      System.out.println("Student not found...");

    em.close();


    System.out.println("done");
```

```
  }
}
```

Update Operation:

Update the marks:

```
public class Main {

  public static void main(String[] args) {

    EntityManagerFactory emf=Persistence.createEntityManagerFactory("studentUnit");

    EntityManager em= emf.createEntityManager();


    Scanner sc=new Scanner(System.in);

    System.out.println("Enter roll to give grace marks ");
    int roll=sc.nextInt();

    Student student=em.find(Student.class, roll); //if it returns the obj then the obj will be in p.state


    if(student == null){
      System.out.println("Student does not exist..");
    }
    else
    {

      System.out.println("Enter the grace marks");
      int marks=sc.nextInt();

      em.getTransaction().begin();

      student.setMarks(student.getMarks()+marks);

      em.getTransaction().commit();

      System.out.println("Marks is graced...");

    }
    em.close();

    System.out.println("done");


  }

}
```

In the above application, we didn't call any update method, we just change the state of the persistence/entity object inside the transactional area, at the end of the transaction, the ORM engine will generate the update SQL.

- This is known as the ORM s/w maintaining synchronization between Entity object and the database table records.
- We have a method called merge() inside the EntityManager obj to update a record also.

Life-cycle of persistence/entity object:-

An Entity object has the 3 life-cycle stages:

1. New state/transient stage
2. Persistence state/managed stage
3. Detached stage

1. New state/transient stage:

If we create an object of persistence class and this class is not attached to the EntityManager object then this stage is known as the new state/transient stage.

example:

Student student=new Student(10,"Ram",780);

2. Persistence stage:

If a persistence class object or Entity object is associated with the EntityManager object, then this object will be in a **persistence stage**.

example:

When we call the **persist(-)** method by supplying the Student entity object then at that time student object will be in a persistence state

OR

When we call the **find()** method and this method returns the Student object, then that object will also be in a persistence stage.

Note:- when an entity class object is in the persistence stage, It will be in-sync with the database table i.e. any change made on that object inside the transactional area will reflect table automatically.

ex:-

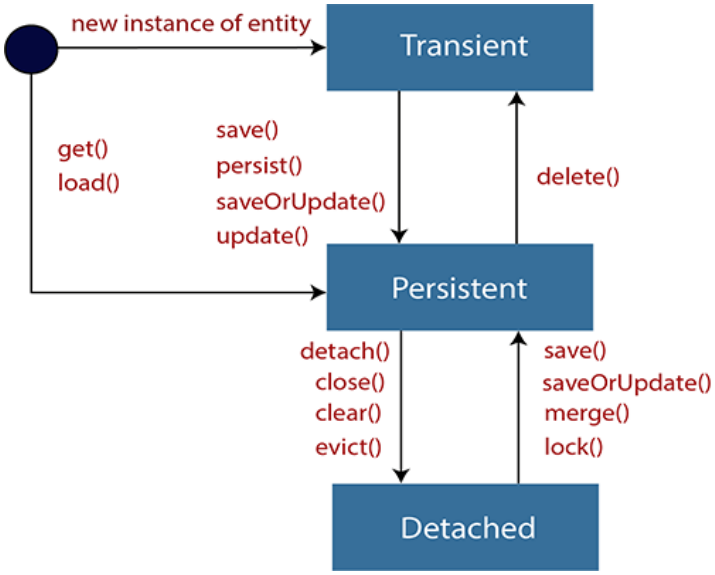
```
Student s=new Student(150,"Manoj",850); // here student obj is in transient state.

em.getTransaction().begin();

  em.persist(s); // here it is in the persistence state

s.setMarks(900);


em.getTransaction().commit();
```

3. Detached stage:

When we call the **close()** method or call the **clear()** method on the EntityManager object, then all the associated entity objects will be in a detached state.

In this stage, the entity objects will not be **in-sync** with the table.

Note:- we have a merge() method in the EntityManager object, when we call this method by supplying any detached object then that detached object will bring back into the persistence state.



Example:

```
//Main.java:-

public class Main {

  public static void main(String[] args) {

      EntityManagerFactory emf=Persistence.createEntityManagerFactory("studentUnit");

      EntityManager em= emf.createEntityManager();

      Student s= em.find(Student.class, 20); //persistence state

      em.clear(); //detached state

      em.getTransaction().begin();

      s.setMarks(500);

      //em.persist(s);// it will throw duplicate ID related exception
      em.merge(s); //persistence state

      em.getTransaction().commit();

      em.close();

      System.out.println("done");
    }
}
```

Note:- To see the ORM tool(Hibernate) generated SQL queries on the console add the following property inside the persistence.xml:

```
<property name="hibernate.show_sql"  value="true"/>
```

To create or update the table according to the entity class mapping information:

```
<property name="hibernate.hbm2ddl.auto"  value="create"/>
```

create: Drop the existing table then create a fresh new table and insert the record.

update: If the table is not there then create a new table, and if the table is already there, it will perform the insert operation only in the existing table.

Some of the annotations of JPA:

@Entity: to make a Java bean class as an entity class, i.e. to map with a table

@Id: To make a field as the ID field (to map with Primary Key of a table)

@Table(name="mystudents"): If the table name and the class names are different

@Column(name="sname"): If the column name of the table and corresponding variable of the Entity class is different.

@Transient: It will ignore the filed value while persisting the Entity object.

@Temporal: To save the Date type of value inside the Database (LocalDate, LocalDateTime)

@Enumerated: We can use the *@Enumerated* annotation to specify whether the *enum* should be persisted by name or by ordinal (default):

Generators in JPA:

Generators are used to generate the ID filed value automatically.

Example:

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int roll;
```

Here roll will be generated automatically for each row.

Note: If we use this **@GeneratedValue** annotation then we should not give the roll explicitly while inserting a record.

So we should create the entity class object by using the zero-argument constructor and set each value by calling the setter method. or we can use an overloaded constructor which ignores the Id field.

For the auto-generated strategy we can use one of the following 3 options:

AUTO: internally underlying ORM s/w creates a table called **"hibernate_sequence**" to maintain the Id value.

IDENTITY: It is used for the auto_increatement feature of the database to auto-generate the id value

SEQUENCE: It is used the **sequence** feature of the database to auto-generate the Id value.

TABLE: Hibernate uses a database table to simulate a sequence.

DAO pattern example with JPA:

```
EMUtil.java:-
----------------

package com.masai.utility;
public class EMUtil {


  private static EntityManagerFactory emf;

  static{
    emf=Persistence.createEntityManagerFactory("account-unit");
  }

  public static EntityManager provideEntityManager(){

    //EntityManager em= emf.createEntityManager();
    //return em;

    return emf.createEntityManager();
  }
}
```

```
Account.java:- (Entity class)
----------------

package com.masai.model;
@Entity
public class Account  {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int accno;
  private String name;
  private int balance;


  public Account() {
    // TODO Auto-generated constructor stub
  }
```

```
  public int getAccno() {
    return accno;
  }


  public void setAccno(int accno) {
    this.accno = accno;
  }


  public String getName() {
    return name;
  }


  public void setName(String name) {
    this.name = name;
  }


  public int getBalance() {
    return balance;
  }


  public void setBalance(int balance) {
    this.balance = balance;
  }


  public Account(int accno, String name, int balance) {
    super();
    this.accno = accno;
    this.name = name;
    this.balance = balance;
  }


  @Override
  public String toString() {
    return "Account [accno=" + accno + ", name=" + name + ", balance="
        + balance + "]";
  }

}
```

```
AccountDao.java:-(interface)
--------------------

package com.masai.dao;

public interface AccountDao {

  public boolean createAccount(Account account);

  public boolean deleteAccount(int accno);

  public boolean updateAccount(Account account);

  public Account findAccount(int accno);

}
```

```
AccountDaoImpl.java:-
---------------------------
```

```
package com.masai.dao;

public class AccountDaoImpl implements AccountDao{

  @Override
  public boolean createAccount(Account account) {

    boolean flag= false;

    EntityManager em= EMUtil.provideEntityManager();

    em.getTransaction().begin();

    em.persist(account);
    flag=true;

    em.getTransaction().commit();


    em.close();

    return flag;
  }

  @Override
  public boolean deleteAccount(int accno) {
    boolean flag=false;


    EntityManager em= EMUtil.provideEntityManager();

    Account acc=em.find(Account.class, accno);

    if(acc != null){

      em.getTransaction().begin();

      em.remove(acc);
      flag=true;

      em.getTransaction().commit();
    }

    em.close();



    return flag;
  }

  @Override
  public boolean updateAccount(Account account) {

    boolean flag=false;


    EntityManager em= EMUtil.provideEntityManager();

    em.getTransaction().begin();

    em.merge(account);
    flag=true;

    em.getTransaction().commit();


    em.close();

    return flag;

  }

  @Override
  public Account findAccount(int accno) {
    /*Account account=null;
```

```
   EntityManager em=EMUtil.provideEntityManager();


    account = em.find(Account.class, accno);




    return account;*/

    return EMUtil.provideEntityManager().find(Account.class, accno);

  }
}
```

persistence.xml:

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

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">


    <persistence-unit name="account-unit" >


<properties>

             <property name="hibernate.connection.driver_class" value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password"  value="root"/>
            <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/ratandb"/>

            <property name="hibernate.show_sql"  value="true"/>
            <property name="hibernate.hbm2ddl.auto"  value="update"/>



        </properties>
    </persistence-unit>
</persistence>
```

```
DepositUseCase.java:-
---------------------------


package com.masai.usecases;
public class DepositUseCase {


  public static void main(String[] args) {

    AccountDao dao=new AccountDaoImpl();

    /*Account acc1=new Account();
    acc1.setName("Ramesh");
    acc1.setBalance(880);



    boolean f= dao.createAccount(acc1);

    if(f)
      System.out.println("Account created..");
```

```
    else
      System.out.println("Not created...");*/


    Scanner sc=new Scanner(System.in);

    System.out.println("Enter Account number");
    int ano=sc.nextInt();

    Account acc= dao.findAccount(ano);

    if(acc == null)
      System.out.println("Account does not exist..");
    else{


      System.out.println("Enter the Amount to Deposit");
      int amt=sc.nextInt();

      acc.setBalance(acc.getBalance()+amt);

      boolean f =dao.updateAccount(acc);

      if(f)
        System.out.println("Deposited Sucessfully...");
      else
        System.out.println("Technical Error .....");

    }
  }

}
```

```
WithdrawUseCase.java:-
------------------------------

package com.masai.usecase;
public class WithdrawUseCase {

  public static void main(String[] args) {

    AccountDao dao=new AccountDaoImpl();

    Scanner sc=new Scanner(System.in);

    System.out.println("Enter Account number");
    int ano=sc.nextInt();

    Account acc= dao.findAccount(ano);

    if(acc == null)
      System.out.println("Account does not exist..");
    else{

      System.out.println("Enter the withdrawing amount");
      int amt=sc.nextInt();

      if(amt <= acc.getBalance()){

        acc.setBalance(acc.getBalance()-amt);
        boolean f=dao.updateAccount(acc);
        if(f)
          System.out.println("please collect the cash...");
        else
          System.out.println("Technical Error...");


      }else
        System.out.println("Insufficient Amount..");
    }
  }
```

```
}
```