TimeStamp Annotations

What is TimeStamp?

TimeStamp is a datatype used to represent(store) date and time

It is commonly used track the event or operation which are performed based on date and time .

Ex:--FlipKart App or Swiggy

Whenever we order food User will not provide the below information .Application is responsible to provide this Like

- Order id (To generate this Generation Strategy is there)-----It Can not be changed
- Ordered time(Odered time and Delivery time has to generated by the application)---It can not be changed

  ordertime remains the same
- Delivery time(ETA---Estimated Time of Arrival)---This depends on some parameters like (Distance + Time Taken To Prepare food)----It can be changed

If I place an order the application should assign the Current virtual machine date and time and it should assign the value to a field(ie ordered time)

*The value will be assigned for Delivery time whenever we update the oder.(ie multiple time)

*LocalDateTime is the Class present in java.time package.

-It represents a date and time without time zone component.

*It is immutable class and thread safe.

LocaDateTime instances are created by factory or helper methods Including

now()----It is a static method which is used to retrieve current date and time

Of()---It is a static method which is used to create the instance date and time classes by specifying specific values for year,month,day,hour,minute,second and nano second.

Comparison with Other Types:

- **LocalDateTime vs LocalDate**: LocalDateTime includes time components (hour, minute, second), while LocalDate represents a date without time.

- **LocalDateTime vs ZonedDateTime**: ZonedDateTime includes a time zone, making it suitable for handling dates and times across different time zones.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Example {
    public static void main(String[] args) {
        // Current date and time
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current LocalDateTime: " + now);

        // Create a specific LocalDateTime
        LocalDateTime specificDateTime = LocalDateTime.of(2024, 7, 15, 12, 30);
        System.out.println("Specific LocalDateTime: " + specificDateTime);

        // Formatting LocalDateTime
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedDateTime = now.format(formatter);
        System.out.println("Formatted LocalDateTime: " + formattedDateTime);
    }
}
```

Whenever we update the data(Distance) there is possibility of changing the delivery time.

But ordered time will not change.

FoodOrder.java(Don’t give the name as Order because Order is the keyword in SQL)

You will get Exception

@Table you can use to give different name

-id (it is order id)

-food_item

-cost

@creationTimeStamp(Both the annotations are from Hibernate not from JPA)

-private LocalDateTime ordered_time;//The value will be assigned only once.

@UpdateTimeStamp

-private LocalDateTime delivery_time;//value will be assigned every time whenever you update the order.

Points:-

-These are the annotations 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.annoatations package.

-Following are the important TimeStamp annotations.

1>@CreationTimeStamp:-

2>@UpdateTimeStamp:-

1>@CreationTimeStamp:-

-This annotation will assign the annotated filed within the entity class with the current virtual machine date and time whenever we save the entity for the first time .

Ie 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.

*The time and date which is inserted into the database server will not be updated or modified afterwards. Eventhough if you update the other fileds of entity .

2>@UpdateTimeStamp:-

This annotation will assign the annotated filed with the current virtual machine date and time when we save or update the entity.

Ie the field which is annotated with @UpdateTimeStamp in the entity class can be updateable later.

(ie When we place an order and when we update the distance by default delivery time will be updated)

Note:-

LocalDateTime:- It is not supported by MySQL 5.5 --→You will get SQLSyntaxException.

If so change the MySQL 5Dialect to MySQL8Dialect in persistence.xml

Note:-If we update the oder(distance,food) ,then the delivery time should change but not ordered time.

To update the order use UpdateOder.java class

To update first you fetch---→Using find() you update it from Biryani to Chicken Biryani

Example Program to understand @CreationTimeStamp and UpdateTimeStamp

@Entity

FoodOrder.java //Hibernate WorkSpace//ZomatoApp

-id

-food_item

-cost

@CreationTimeStamp(Ordered time will not change)

-private LocalDateTime ordered_time;

@UpdateTimeStamp(delivery time would update)

-private LocalDateTime delivery_time;

//Getters and Setters//

PlaceOder.java

main()

EMF(If you give persistence unit name wrongly you will get PersistenceException

EM

ET

FoodOrder order=new FoodOrder();

order.setFood_item(“Biryani”);

order.setCost(150);

manager.persist(order);

UpdateOrder.java

```
  main()
      EMF
      EM
      ET
//Before you update first you fetch
FoodOrder oder=manager.find(FoodOrder.class,1));
//Update the food_item
  Order.setFood_item(“Panner Biryani”);
transacation.begin();
transaction.commit()
```

Assignment Questions:-

1>Find Food order by id

find(FoodOrder.class,1)

2>Find FoodOrders by food item

select f from FoodOrder f where f.food=?1

Use getResultList();

3>Find FoodOder by id and food

select f from FoodOder f where f.id=?1 and f.food=?2

Use getSingleResult()

4>Find Food Orders by cost

select f from FoodOrder f where f.cost=?1

5>Filter food orders between a range of cost.

select f from FoodOder f where f.cost between ?1 and ?2

```
import java.time.LocalDateTime;
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

@Entity
public class FoodOrder
{
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String food_item;
	private int cost;

	private LocalDateTime deliverytime;

	@CreationTimestamp
	private LocalDateTime  ordered_time;

	@UpdateTimestamp
	private LocalDateTime OrderUpdateTime;
```

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

EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man=fac.createEntityManager();
EntityTransaction tran=man.getTransaction();
   tran.begin();

   FoodOrder od=new FoodOrder();
   od.setFood_item("Pulav");
   od.setCost(50);
//To get the current system date and time
   LocalDateTime now = LocalDateTime.now();
//add 20 minutes to the current time that will be the delivery time
   LocalDateTime deliverytime=now.plusMinutes(20);

   od.setDeliverytime(deliverytime);
   man.persist(od);
   tran.commit();
}
}
```

```java
public class UpdateOrder {
public static void main(String[] args)
{
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter order id to update");
    int oid=sc.nextInt();

    EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
    EntityManager man=fac.createEntityManager();
    EntityTransaction tran=man.getTransaction();
    tran.begin();
    FoodOrder od=man.find(FoodOrder.class, oid);
     if(od!=null)
         {
     od.setFood_item("RiceBath");
     od.setCost(185);//Not Mandatory
     LocalDateTime updateddeliverytime=LocalDateTime.now().plusMinutes(20);
//It is mandatory to update delivery time whenever you update the food
     od.setDeliverytime(updateddeliverytime);//od.setDeliverytime(LocalDateTime.now().plusMinutes(20));

     tran.commit();
         }
     else
        {
     System.out.println("Order Can not be updated since order id is wrong");
        }
```

Composite Key

The PK is used to identify the record uniquely.

The combination of two or more columns to form a PK is known as Composite Key

Composition means combining more than one thing

In a table it is always recommended that only one PK must be there

It is a combination of two or more columns (fields) that together uniquely identify a record in a table.

Each combination of values in the composite key must be unique within the table.

This means no two rows can have the same combination of values across all columns that make up the composite key.

In Primary key -----column means single column but

Composite Key------ Column means multiple columns

When we can use this?

Composite keys are often used when no single column can uniquely identify a record, but the combination of several columns can.

Take One Example where one column is not enough to identity the record uniquely in the table

Lets consider One Training Institution -----BTM Jspiders-----→One DataBase as BTM
In BTM Databse Lets consider 3 tables

Student Table

sid	sname
1	Guru
2	Raj

Course Table

cid	cname
101	J2EE
102	Java

Master Table

sid	sname	cid	cname
1	Guru	101	J2EE
2	Raj	102	Java
1	Guru	102	Java
2	Raj	101	J2EE

Here the combination of sid and cid forms composite key

If you want to save the data in Master Table then you Should create an entity class (Master) like bleow
Now Our Master Class is Composite key class .
It is a rule that Composite Key class must implement Serializable but it violets the rule of POJO

@Entity
class Master implements Serializable
@Id
-sid
@Id
-cid
-sname
-cname Output

```
create table Master (
       cid integer not null,
        sid integer not null,
        cname varchar(255),
        sname varchar(255),
        primary key (cid, sid)
    ) engine=MyISAM
```

It is not recommended to have 2 PK's in a Entity class for Below Reasons

- **Normalization**: Having two separate primary keys implies that each one independently identifies a unique record, which can violate normalization principles. Proper normalization usually involves ensuring that a table has a single, unique primary key.

- **Referential Integrity**: Foreign key constraints and other relational database features are designed to work with a single primary key. Having multiple primary keys would complicate these relationships and potentially lead to data integrity issues.

```
import java.io.Serializable;
import javax.persistence.Embeddable;

@Embeddable
public class MasterId implements Serializable
{
      private int sid;   //Embeddable Class which includes the  all the fields  which are responsible for composite Key
      private int cid;  //No need of @Id

        //Setters nad Getters
       //Override toString()
}
```

Note:-If you don't implement this MasterId class with Serializable then you will get PersistenceException with a root cause MappingException

```
Message:-Composite-id class must implement Serializable
```

Note:-Hibernate WorkSpace/CompositeKeyFinal

```java
@Entity          //Don't Forget this
public class MasterStudentInfo
{
private String sname;
private String cname;
@EmbeddedId                 //In this line it shows error ignore it //It will force you
private MasterId mid;       //to override equals() and hashcode()//Not Mandatory
//Above Declare the variable of MasterId
//  Setters and Getters  //
//       Override toString()  //

}
```

Composite key instances are often used in collections such as HashSet or as keys in HashMap . These collections rely on equals() and hashCode() to determine object equality and to manage object storage efficiently.

```java
public class SaveMasterStudentInfo
{
public static void main(String[] args) {
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man=fac.createEntityManager();
EntityTransaction tran=man.getTransaction();
tran.begin();
MasterId mi=new MasterId();
mi.setSid(2);
mi.setCid(102);

MasterStudentInfo msi=new MasterStudentInfo();
msi.setSname("Raj");
msi.setCname("Java");
msi.setMid(mi);//
man.persist(msi);
tran.commit();
```

```
public class FetchStudentInfo {
public static void main(String[] args) {
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man=fac.createEntityManager();

MasterId mid=new MasterId();//Composite Key Class
//Find the Student info based on Composite Key
mid.setSid(1);
mid.setCid(101);

MasterStudentInfo msinfo=man.find(MasterStudentInfo.class, mid);
if(msinfo!=null)
{
System.out.println(msinfo);

}
else
{
System.out.println("either student id or course id is wrong");
}
```

Composite Key

If you want to give the PK column by combining more than one column then we use Composite Key concept

Ex:-Lets consider User table

Id|name|phone

-Here if I make the id column as PK column then duplicates will not be allowed here

-If I make the phone column as PK column again duplicates will not be allowed here.

Lets consider User table

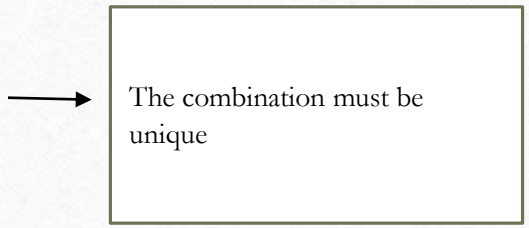
Name | Phone|Email--→Composite Key

In case of Composite Key

The combination of more than one column must be unique in each record .

User

Name	Phone	Email
Xyz	888	xyz@gmail.com
Xyz	888	abc@gmail.com



Note:-

(Composite key class must implement Serializable)

Why the composite key class should implement Serializable?

By implementing Serializable, you ensure that instances of **CompositeKey** can be serialized and used in scenarios where serialization is required, enhancing the flexibility and compatibility of your Hibernate-based application.

Implementing Serializable allows instances of the composite key class to be serialized, which is essential for various use cases, such as caching, distributed computing, or passing objects between different layers of an application.

```
@Entity
public class User implements Serializable
    @Id
      private int id;          //No this class contains 2 PK's
    @Id
      private String phone;
      private String name;
      private String password;
```

There are 2 different way in which we can create composite key

Note:-

1st way:-

We need to make the class which includes the composite key as Serializable

The Composite Class must implement Serializable

And 2 Primary keys in one class is not recommended

The first way would violets the rule of POJO since the pojo class must not implement any interface.

But here it is implementing Serializable interface

2nd way:-

Write one class(MasterId), there have all the attribute of MasterStudentInfo class which are responsible to create composite key(ie email and phone)

Now the MasterId class will be embedded class it should implements Serilaizable.(Use @Embeddable here)

Embedded class means class is fixed or Embedded into another class

Points:--

- The combination of more than one column to form a PK(Primary Key) is called as Composite Key.

Note:-

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

The below table represents the working of Composite Key

Composite Key

Email	phone	name	password
abc@gmail.com	888	ABC	abc123
abc@gmail.com	888	XYZ	xyz123
abc@gmail.com	777	XYZ	abc123
pqr@gmail.com	777	ABC	xyz123
pqr@gmail.com	777	XYZ	abc123

These two combination should not repeat

Points:

@Embeddable

-It is a class level annotation belongs to JPA and present in javax.pesistence package.

@Embeddable annotation is used to indicate that a class can be embedded(fit) within an entity

-It is used to mark the class as Embeddable class.

-An Embeddable class will have all the fields which are responsible for the creation of Composite Key.

-An embeddable class must implements 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 filed as Embedded id(Composite Key)

-The field which is annotated with @Embeddable must be of Serializable type else we will get exception.

Demo:-

Class User implements Serializable

-name

@Id

-phone

@Id

-email

-password

If I don’t use serializable

we will get MappingException

Don’t use generated value

-If I implement we will violet the rule of POJO

+class Test

EMF

S.O.P(factory);

Example program to understand @Embeddable and @EmbeddedId

@Embedddable
+ class UserId implements Serializable

-phone
-email

] these two attributes of User class IS contributing to the composite key so write here not in user

@Override
toString()

@Entity

User.java

-name

-password

@EmbeddedId

private UserID userid;

//To support POJO rule in user class use this

//Getters and Setters

//toString()//

SaveUser.java

```
 main()
     UserId userid=new UserId();
                userid.setEmail("xyz@gmail.com");
                 userid.setPhone(9999);
 User user=new User();
user.setName("ABC");
user.setPassword("abc1234");
user.setUserId(userId);
EMF
EM
ET
```

```
manager.persist(user);
tran.begin();
```

FetchUserByPrimaryKey.java

```
 main()
   UserId userid=new UserId();
   userid.setEmail("abc@gmail.com");
userid.setPhone(888);
EMF
EM
User user=manager.find(User.class,userid);
If(user!=null)
   S.O.P(user);
```

else

S.O.P(Invalid phone number or email)

Output:-

User[name=ABC,password=ABCD1234,userId[phone=9999,email=xyz@gmail.com]

Hibernate LifeCycle

How Human beings are having our own life cycle .Butterfly is having its own life cycle

Is there any lifecycle for Hibernate objects means --->yes

Previouly I told you about transient State,Persistent State,etc still some more states are there

What is the difference between each state and the task performed by an object in each state is different

In Hibernate we have saved the record ,we have updated ,deleted ,fetched ,all crud operations we have performed

-Whenever we create object by using new keyword then the object will be there in transient state.

-Whenever it is connected with session or EntityManager then it will enter into Persistent state.

-When the object goes from Transient State to Persistent State means ?

If we call save()or update() or saveOrUpdate() or persist() or merge()

-In transient State object will not represents any record in the table any modification will not affect the record

-The object in persistent state will represents a record in the table

We were using find() or get() to fetch the record -->That is Persistent State

-If the object is connected to the session then it is said to be Persistent state.

- If the object is disconnected from the session –it will go to the Detached state.
- If you close EntityManager or Session it goes to detached State or if you call detach() method the object in the persistent state will go to detached state.
- -Is there any possibility to move the object which is there in Detached State to Persistent State?---yes by calling find()

Note:-

Once we close the program or Execution of the Program Completes

Garbage collector will remove all the object from the heap ie it removes the implementation class object of Entity Manager from heap that means EntityManager is going to close.

Now the record is there in the database but it is not connected with Session

-

How To move the object from detached State to Persistent state ?For that we can use below methods

get()

load() or find()

-If we change the values of Transient object it will not affect the record in the table.

When do we call an object is in detached State?

When object is disconnected from the Session.

We have one More State of Hibernate Object ie RemovedState

When the object goes to removed State?

Whenever we delete an object it goes to removed State. To delete a Persistent object

we have we can use delete() or remove()

delete() can delete both Persistent object and Detached Object

but remove() can delete Only delete persistent object //

So if you try to delete using detached object using remove() -→you will get IllegleArgumentException

So What are the States of an object in Hibernate LifeCycle

-Whenever we use delete() or remove() then the object will go to Removed State

-In Detached State object will be there in Database but not connected with Session

-We can not delete the object which is there in detached State because it is not connected with session.

-find() is used to move the object from detached state to persistent state.

Points:-

-Hibernate LifeCycle represents the different states of an object in Hibernate.

- Following are state of an object in Hibernate LifeCycle.

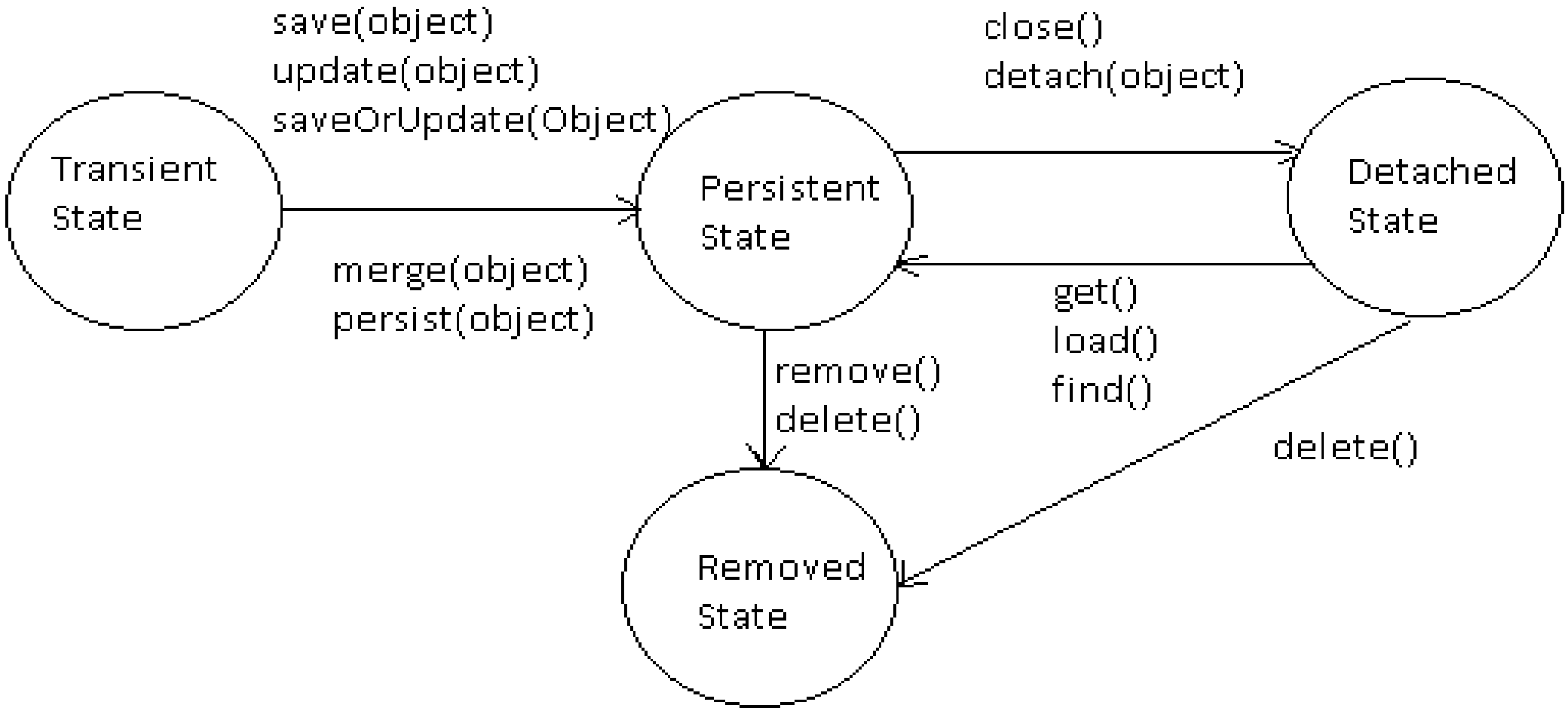
1>Transient State

2>Persistent State

3>Detached State

4>Removed State

Hibernate LifeCycle



1>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 any effect on the record in the table.

2>Persistent State

-An object is said to be in persistent state once it is connected with the Session.

-An object in the persistent state will represents a record in the database server.

-Any modification on the State of Persistence object will directly affect the record in the table.

3>Detached State:-

-An object is said to be in detached state once it is disconnected from the session(Entity Manager).

-An object in the detached state will not represent any record in the database server.

So modification of detached object will not affect any record in the database server.

4>Removed State:-

-An object is said to be in removed state once it is deleted from the database server.

-We can not delete an object in the detached State by remove(object)

-remove(object) can only delete persistent object.

-delete(Object) can delete the object which is present in Persistent and Detached State

HibernateLifeCycle_Proj

Example Program to understand Hibernate LifeCycle

```
public class TestHLC
{
public static void main(String[] args)
{
User u=new User();//Transient State//Not Representing any record
u.setName("Guru");
u.setEmail("guru@gmail.com");
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man=fac.createEntityManager();
EntityTransaction tran=man.getTransaction();
tran.begin();
man.persist(u);//Persistent State//Represents a record
tran.commit();
```

```
tran.begin();
u.setName("Raj");
    u.setEmail("raj@gmail.com");
   tran.commit();//While Updating delete the old database//Single Record must be there
   tran.begin();
   man.detach(u);//Detached State//Not Representing any record
   tran.commit();
   tran.begin();
   u.setName("Rakesh");
   u.setEmail("rakesh@gmail.com");
   tran.commit();//Any Modification on Detached State object will not affect the record

tran.begin();

man.remove(u);//Object is there in detached State// You will get Exception in thread "main"
java.lang.IllegalArgumentException: Removing a detached instance
   tran.commit();



}

}
```

Note:-

-Here if 1 is not present then find() will return null .Now p will hold null

```
public class RemoveUser
{
public static void main(String[] args) {
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man=fac.createEntityManager();
EntityTransaction tran=man.getTransaction();
tran.begin();

User u=man.find(User.class, 1);
if(u!=null)
{
man.remove(u);//Persistent Object will be removed else It will print
IllegleArgumentException(if the User info is not present for particular id)
tran.commit();
}
else
{
System.out.println("No user found");
}
}

}
```

//Transient Object is passed to the remove() below it will throw IllegaleArgumentException

```
User u=new User();
u.setId(1);
u.setName("Ram");
u.setEmail("ram@gmail.com");
```

-If I directly use it will be

```
manager.remove(u);//Now it will throw IllegleArgumentException
```

When you will get NullPointerException?

```
u.setAge(25);
```

U is null so any operation on null reference throws an exception called NullPointerException.

So use if(u!=null)

Cache Mechanism

It is used to store copies of frequently accessed data from the main memory (RAM) to reduce latency and improve performance.

- Cache Mechanism is use to improve the performance of an application.

-Lets consider there is no concept of Cache.

-If I want to fetch the data of an user from the database server where id=1 then SQL query has to be executed.

If you don't use ,the traffic increases between java application and the database server.

How will you improve the performance of an application?

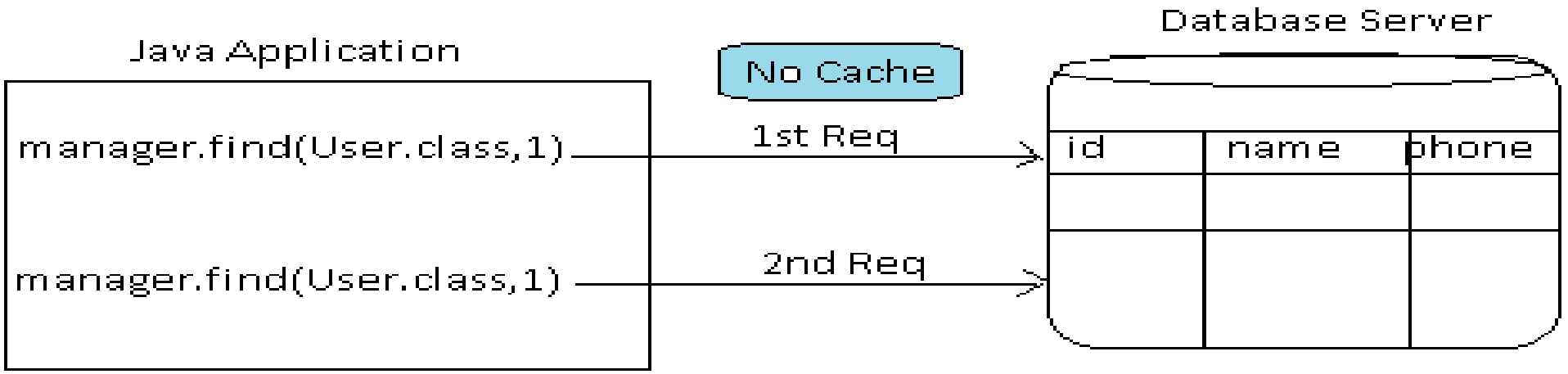
By Using cache mechanism

*By default hibernate supports First Level Cache.

*2nd Level cache is not enabled

*Hibernate allocates 1st level cache for each Session.

Lets consider User table



Here the traffic increases hence the efficiency decreases
Response time will also increases
But this will not happen in Hibernate Every EntityManager/Session will associated with cache memory
If you create 10 session all 10 session will have cache memory.----That is called as First Level Cache
Hibernate Supports 2 levels of cache
1st Level Cache
2nd Level Cache
Entire Hibernate Application will have one Second Level Cache Memory which is connected to 1st Level Cache Memory

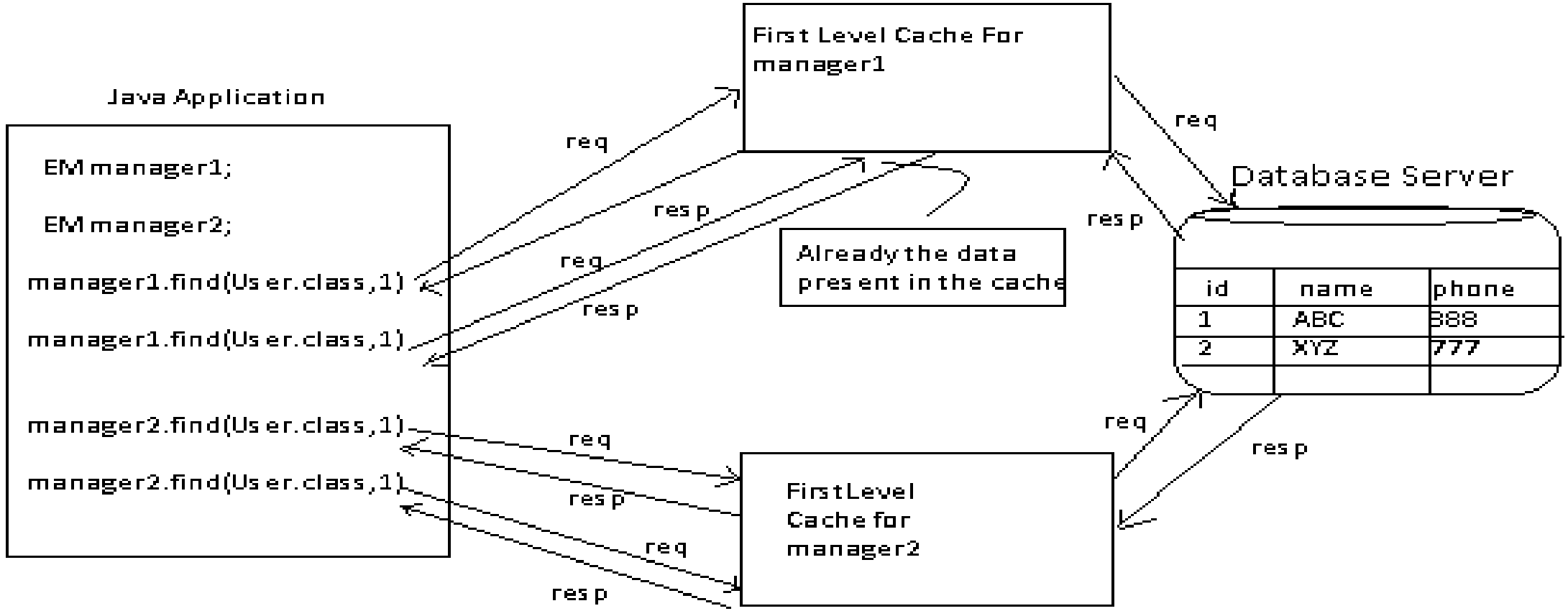
For Time being lets consider 2 EntityMangers
EntityManager1 and EntityManager2
2 EntityManagers are nothing but 2 Sessions
In this case we will have 2 First Level Cache Memory and by default 2nd level cache memory is not enabled
By default 1st level cache is enabled
2nd level cache has to be enabled explicitely

manager1.find(User.class,1)----Now the request made from the Java Application will go to the 1st Level cache memory which is allocated to EntityManager 1
-Now is there any object present in the 1st Level Cache memory With respect to the user ----No
-Now It will hit the database server --→If the user info present, an object of User is going to create and a copy of that object is going to store in the 1st Level Cache to serve future request.Finally It will be given back to our Java Application.
-Now I will write manager1.find(User.class,1)-----Now request will go to 1st Level Cache --→User 1 is present it will serve the request there it self--→It will not hit the database server.
-What if 1st Level cache is not there ,how many request will go to the database server?----2 Requests

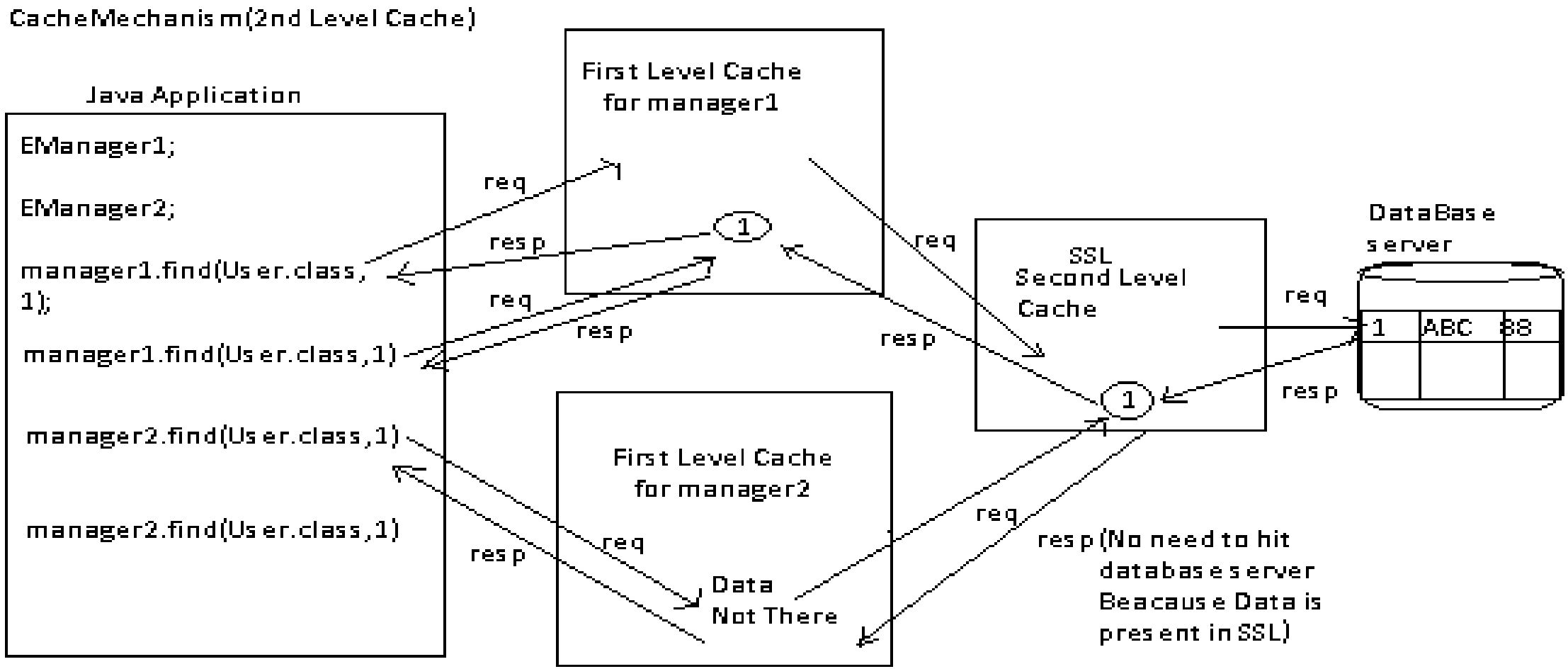
Now manager2.find(User.class,1)--→Now tell me where the request will go ?---
Because of this Eventhogh we made 4 requests only 2 requests are hitting the database server.

Explain Second Level Cache.
Previously 2 Requests were hitting the Database server but now Using second level cache only one request will hit the database server.

The usage of 1st level cache



Usage of SecondLevel Cache



For 2 EntityManager ----2First Level Cache Will be provided by Hibernate.

Note:-

To enable the 2nd Level cache ---→we need to add one dependency

i e EHCache Relocation in pom.xml

-The Version of EHCache relocation and Hibernate Core-Relocation must be same

-Just go to the git hub link https://github.com/GuruJSP

Copy <shared-cache-mode> tag and paste it in persistence.xml of your program inside <persistence-unit>

Then add <property name=”hibernate.cache.use”>

<property factory class>

Annotate your entity class with @Cacheable

Points:-

Cache:-

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

-Hibernate Supports 2 levels of cache mechanism using which we can reduce the traffic between the Java application and the database server and increases the performance.

-Following are the 2 Levels of Cache Supported by Hibernate

1>First Level Cache

-It is a layer of storage assigned by the hibernate for every session(EntityManger) 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.

2>Second Level Cache

- It is a layer of storage which is shared by all the sessions(Entity Managers) created by a SessionFactory(EntityMangerFactory).

- By default only First Level cache is supported by Hibernate and the 2nd level cache has to be enabled explicitely.

Steps to enable 2nd Level Cache

Step1:-Add the hibernate EHCache Relocation dependency in pom.xml

Step2:-Add the following code within <persistence-unit>

Just above <properties> tag

<shared-cache-mode>Enable_SELECTIVE</shared-cache-mode>

Note:<shared-cache-mode>Enable_SELECTIVE</shared-cache-mode>, it means that the second-level cache is enabled only for those entities that are explicitly marked as cacheable

Step 3:-Add the following properties in persistence.xml

<proprty name="hibernate.cache.use_second_level_Cache" value="true"/>

<property name="hibernate.cache.region.factory_class" (To Store) value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>

-The property hibernate.cache.region.factory_class in your Hibernate configuration specifies the

class that Hibernate should use to manage the second-level cache regions.The value org.hibernate.cache.ehcache.EhCacheRegionFactory indicates that Hibernate should use EHCache as a Caching Provider.

Step 4:-Annotate your entity class with @Cacheable

Program To Understand Cache-Mechanism

@Cacheable

User

-id

-name

-phone

id	name	phone
1	Guru	9483663883
2	Raj	9482205408

//Setters and Getters

//Override toString()

```java
public class FindUser1
{
public static void main(String[] args)
{
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man1=fac.createEntityManager();//one 1st Level cache for man1
EntityManager man2=fac.createEntityManager();//one 1st Level cache for man2 //Toatally 2 First –Level caches will be given


//Without using Second Level Cache if you execute you totally 4 times it hits the database server
//2-1st Level caches are going to create for man1 and man2 here
//Here 2 --1st level caches will be there but not 2nd level cache
man1.find(User.class, 1);//It Hits the database server
man1.find(User.class, 2);//It Hits the database server                    // Without Using 2nd Level Cache
                                                                          //It will hit the database server 4 times
man2.find(User.class, 1);//It hits the database server
man2.find(User.class, 2);//It hits the database server

//Since the data is already present in 2-1st level caches again it will not hit the database server.

man1.find(User.class, 1);//It will not hit the database server
man1.find(User.class, 2);//It will not hit the database server

man2.find(User.class, 1);//It will not hit the database server
man2.find(User.class, 2);//It will not hit the database server

}

}
```

```java
public class FindUser2
{

public static void main(String[] args)
{
EntityManagerFactory fac=Persistence.createEntityManagerFactory("dev");
EntityManager man1=fac.createEntityManager();
EntityManager man2=fac.createEntityManager();

//Here in this code Since we have enabled 2nd level cache only 2 time it will hit the database server
//2 times it will hit the database server
man1.find(User.class, 1);//It will hit the database server,object is going to store in 1st and 2nd level cache
man1.find(User.class, 2);//It will hit the database server,object is going to store in 1st and 2nd level cache
//It will not hit the database server
man2.find(User.class, 1);//Nothing is there in 1st Level cache but object is present in 2nd level cache
man2.find(User.class, 2);//Nothing is there in 1st Level cache but object is present in 2nd level cache
}

}
```

//After enabling 2nd Level cache
//only 2 times it will hit the database server

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">
<persistence-unit name="dev">
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>      <!-- shared-cache-mode: This element specifies the caching mode to be used -->
<properties>
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/CacheMechanismDb2?createDatabaseIfNotExist=true"/>
<property name="javax.persistence.jdbc.user" value="root"/>
<property name="javax.persistence.jdbc.password" value="admin"/>
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>

<!-- for caching -->
 <property name="hibernate.cache.use_second_level_cache" value="true"/>//Directly  it can be enable or dissable
 <property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>

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

Note:-

<property name="hibernate.cache.use_second_level_Cache" value="true"/>

Here if you make value="false"------→Totally 4 Queries are going to display on console window ----→Since you are not using 2nd Level Cache

Here if you make value="true"------→Totally 2 Queries are going to display on console window ---→Since we are using 2nd Level cache

```
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
```
<!-- **shared-cache-mode**: This element specifies the caching mode to be used -->

That is the Entity class which is annotated with @Cacheable will be involved in the cache mechanism not any other class.

Common Values for <shared-cache-mode>:

1.ALL:

- **Description**: This mode enables caching for all entities, meaning every entity managed by the persistence context will be cached.
- **Use Case**: Suitable for applications where almost all entities are frequently accessed and can benefit from caching to improve performance.

2.NONE:

- **Description**: Disables caching entirely. No entities will be cached.
- **Use Case**: Useful for scenarios where caching is not desired, such as applications where data changes frequently and cache consistency is hard to maintain, or where memory constraints are strict.

3.ENABLE_SELECTIVE:

- **Description**: Enables caching only for entities explicitly marked with the @Cacheable annotation.
- **Use Case**: Ideal for applications where only specific entities benefit from caching. It allows fine-grained control over which data is cached.

4.DISABLE_SELECTIVE:

- **Description**: Enables caching for all entities except those explicitly marked with an annotation (e.g., @Cacheable(false)).
- **Use Case**: Useful when most entities benefit from caching, but a few specific entities should not be cached due to their nature or the need for real-time data consistency.

The hibernate.cache.region.factory_class property in Hibernate is used to specify the cache provider and the strategy for managing the second-level cache regions. The value org.hibernate.cache.ehcache.EhCacheRegionFactory indicates that EhCache is being used as the second-level cache provider.

EhCache is an open-source, Java-based caching library provided by Terracotta, a division of Software AG.

Some other cache provider

Hazelcas

Infinispan

Redis

Association Mapping Summary

*In oneToOneUni Mapping -→ Totally 2 tables are going to create-→1 table will have FK

*In Bi-Directional Mapping owning side and non owning side will come into picture.

Note:-

In Maven Project if you create 2 packages and in each package if you try to create same class Maven will not allow you will get Exception----DuplicateMappingException
The [org.jsp.onetoonebi.PanCard] and [org.jsp.associationpractice.PanCard] entities share the same JPA entity name: [PanCard] which is not allowed!

*In oneToOneBi-Directional Mapping 2 table are going to create but both the table will have FK which is not required so we need to use owning side and non owning side

*While writing the code in OneToOneBi-Directional Mapping
p.setCard(card);
//card.setPerson(p);//If you don't use this line then in PanCard table person_id(FK) will be null
But you will not get any exception.

Use cascade attribute to avoid writing multiple persist( );

*OneToManyUni Ditectional mapping ------→3 Tables are going to create but the 3rd Table Contains the PK's of first two tables..
We can not avoid the creation of 3rd table in oneToManyUni
*3rd Table is called as JoinTable ----Which is used to build the relationship
*when you do not use a direct foreign key in the child entity. This approach ensures that the relationship is managed via a separate table,

Explanation:-

```
@JoinTable(    name = "Department_Employee",
 joinColumns = @JoinColumn(name = "departmentId"),
 inverseJoinColumns = @JoinColumn(name = "employeeId"))
```

•**name = "Department_Employee"**: Specifies the name of the join table.
•**joinColumns = @JoinColumn(name = "departmentId")**: Defines the foreign key column in the join table that references the primary key of the owning entity (Department)
•**inverseJoinColumns = @JoinColumn(name = "employeeId")**: Defines the foreign key column in the join table that references the primary key of the target entity or Non –Owining Entity(Employee).

•**name = "Department_Employee"**: Specifies the name of the join table.
•**joinColumns = @JoinColumn(name = "departmentId")**: Defines the foreign key column in the join table that references the primary key of the owning entity (Department)
•**inverseJoinColumns = @JoinColumn(name = "employeeId")**: Defines the foreign key column in the join table that references the primary key of the target entity or Non –Owining Entity(Employee).

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

1>Creational Design Pattern

2>Structural Design Pattern

3>Behavioral Design Pattern

1>Creational Design Pattern

It is used to write the object creation logic

Ex:-

- Singleton Design Pattern
- Factory Design Pattern
  - -beginTransaction( ),getTransaction( ),createEntityManager( ) etc..

2>Structural Design Pattern

To devide the application into

Presentation Logic

Persistence Logic

Business Logic we use this design pattern

Ex:-

DAO----Data Access Object

The main purpose of a DAO is to separate the data access logic from the business logic in an application,

DTO----Data Transfer Object

DTOs are simple objects that contain data fields to transfer data between software application subsystems, layers, or services.

3>Behavioral Design Pattern

Ex:-Observer design pattern:-

Weather Forecasting Application(It will notify the Subscriber about current weather updates)

Advantages of Using Design Pattern

-We can re-use the code.

-Code optimization

--Code readability increases

-To achieve code Modularity

In JPA we were writing to perform any operations like save,update,delete , fetch we were writing separate classes
Now we will write all these logics in DAO
-Controller is used to handle the request
-Inside DTO --→generally we write entity classes .

Points :-

-Design pattern are used to provide the solution for various technical problems that we might face during the application development.
-By using design pattern we can achieve code modulazition, code-reusability and also we can optimize the code.
-Following are the important design pattern:-

1>Creational Design Pattern
-Singleton Design Pattern
-Factory Dessign Pattern
2>Structural Design Pattern
-DTO(Data Transfer Object)
-DAO(Data Access Object)
-MVC(Model View Controller),etc…

3>Behavioral Design Pattern

Observer Pattern:-This pattern defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Requirement

Create the following entity classes

1>Merchant(id, name, phone, gst_number, email and password)

2>Product(id,name,category,brand,description,cost and image url)

Note:-Merchant and Product have OneToMany Bi-Directional relationship.

Perform the following tasks

1>Save Merchant

2>Update Merchant

3>Find Merchant by id

4>Verify Merchant by email and Password

5>Verify Merchant by Phone and Password

6>Add Product

7>Update Product

8>Find Products by Merchant id

9>Find Products by Brand

10>Find Products by category

Project Structure

Merchant_Product_App

src/main/java

|

|

---------- org.jsp.merchantproductapp.controller(Package For Controller)

| |

| ---------- MerchantController.java

|

|

|

---------- org.jsp.merchantproductapp.dao

| |

| ---------- MerchantDao.java

|

----------org.jsp.merchantproductapp.dto

|

| ---------- Merchant.java

| ---------- Product.java

src/main/resources

META-INF(Folder)

----persistence.xml

Steps:

1>Create a Simple Maven Project (Merchant _Product_App)

----Add the dependencies(Hibernate Core-Relocation and Mysql Connector.jar)

2>Create persistence.xml-----in META-INF Folder--→which has to be created in src/main/resources

3>Create 3 Packages

- org.jsp.merchantProductApp.dto
- org.jsp.merchantProductApp.dao
- org.jsp.merchantProductApp.controller

4>Create 2 Entity Classes under dto

1>Merchant.java(id,name,phone,email,gst_number,password)

Use mappedBy here

```
@OneToMany
List<Product> products;
```

2>Product.java(id,name,brand,category,description)

Use JoinColumn here

```
@ManyToOne
Merchant merchant;
```

5>Because of this 3 tables will be created so use mappedBy and JoinColumn

Create a DAO Class

MerchantDao

Write EMF and EM in class block --→instead of writing it in each methods

Don't write ET in Class block

write saveMerchant( )---→return type is Merchant-→Because ,we need to print the id of Merchant once after saving it.

In updateMerchant( )---→Fetch it from DB---→And update it.

If the record is not present it will return null

findMerchantById( )---→Wri

In Controller Package

-MerchantController.java

main()

First Write all the 10 SOP Statements

Declare Scanner class outside the main method by making it Static

Switch and Case Statement inside main method---Just below the SOP

Case 1:saveMerchant()-----→Case1 calls saveMerchant()

Now it will ask you to create one static method create it--→Write the logic inside it

Inside this method you need to ask the user to enter the Merchant info ---→Since it is in Controller class

Like Below

```
Merchant m=new Merchant();
m.setName(sc.next());
```

Now we need to pass this Merchant object to the saveMerchant(Merchant m)method which is there in MerchantDao class

But before that We need to create MerchantDao Class

Now create MerchantDao clas------goback to MerchantController class

Create an object of MerchantDao in Merchant Controller class----Write it in class level by making it static

Since we need to use it in static method ie----→saveMerchant() within Merchant Controller.

Now inside saveMerchant() -----within Merchant Controller class--→

and write

```
mdao.saveMerchant(m);//MerchantDao is declared outside the main method as static
```

Now the below line will force you to create a method in MerchantDAO

```
mdao.saveMerchant(m);
```

Now saveMerchant() is going to create in MerchantDao
Here in this class just declare the below things above of all methods

```
EMF
EM//Since these two are used in all the methods //To avoid duplication
```

Then go to saveMerchant(Merchant m)---→Write

```
ET
man.persist(merchant)
tran.commit();
return m;
```

Now go to MerchantController class -----saveMerchant() method----
Write the below Statement

```
m=mdao.saveMerchant(m);//take the saved merchant back
System.out.println("Merchant is registered with id "+m.getId());
```

Finally Execute MerchantController Class which contains main method

Next for Update Merchant

Go to First you go to Merchant Controller class

We have SwitchCase ---there ---------Write Case2-----ie updateMerchant() like below

```
case 2:
updateMerchant();
break;
```

Now it will ask you to create a static method called updateMerchant()
Create it
Now inside this method
Ask the user to enter merchant id(mandatory)
Fields which you want to update (name,email,password)etc
Not mandatory to provide all the fields
Since we are passing this Merchant object which contains all the fields which are necessary for updation to the MerchantDao class---so id is mandatory

```
System.out.println("Enter Merchant id to update and other fields(not all --some)");
Merchant m=new Merchant();
m.setId(sc.nextInt());
m.setName(sc.next());
```

```
m.setEmail(sc.next());
m.setPassword(sc.next());
mdao.updateMerchant(m);
```

Now the above line will force you to create a method updateMerchant(m) in MerchantDao class

Select the option -→so that it will create a method in MerchantDao class
In MerchantDao class within ---→updateMerchant() method write below code

```
____EntityTransaction tran=man.getTransaction();
tran.begin();
Merchant dbm=man.find(Merchant.class, merchant.getId());
if(dbm!=null)
{
dbm.setName(merchant.getName());
dbm.setEmail(merchant.getEmail());
dbm.setPassword(merchant.getPassword());
tran.commit();
return dbm;
}
```

```
else
{
return null;
}
```

We need to return --→To write the SOP to display user entered id is wrong in the Merchant controller like below

```
Merchant mdb=mdao.updateMerchant(m);//
if(mdb!=null)
{
System.out.println("Merchant is Updated with an id "+mdb.getId());
}
else
{
System.out.println("Merchant not found since you have enetered wrong id to find");
}
```

Next for findMerchantById

Same steps as previous

Case 3:findMerchantById()

Ask the user to enter id not

call findMerchantById(mid) which is present in MerchantDao class

In MerchantDao Class

```
public Merchant findMerchantId(int mid)
{
 return man.find(Merchant.class, mid);

}
```

In MerchantController Class

Write this below

```
private static void findMerchantById()
{

System.out.println("Enter Merchant id to find");
int mid=sc.nextInt();
Merchant m=mdao.findMerchantId(mid);
if(m!=null)
{
System.out.println("Merchant found");
System.out.println(m);
}
else
{
System.err.println("No Merchant found for the entered id");
}

}
```

Next for findMerchantByEmailAndPassword

```
private static void verifyMerchantByEmailAndPassword()
{
System.out.println("Enter Email id");
String email=sc.next();
System.out.println("Enter Password");
String password=sc.next();

Merchant m=mdao.findMerchantByEmailAndPassword(email,password);
if(m!=null)
{
System.out.println(m);
System.out.println("Merchant is verified");

}
else
{
System.err.println("Merchant email/password is invalid");
}
}
```

```
public Merchant findMerchantByEmailAndPassword(String email, String password)
{
     Query q=man.createQuery("select m from Merchant m where m.email=?1 and
m.password=?2");
     q.setParameter(1, email);//Don’t pass the values to the above query directly
     q.setParameter(2, password);//Use this setParameter else you will get error

     try {
          Merchant m=(Merchant) q.getSingleResult();
          return m;
} catch (NoResultException e)
     {
         return null;

 }
}
```

Similarly findMerchantByPhoneAndPassword