Day5: Has-A Mapping, IS-A mapping, Cascading, Fetch type

Mismatched between Object-Oriented Representation and relational representation of data:

1. Granularity mismatch:- HAS-A relationship problem
2. Inheritance mismatch :- IS-A relationship problem
3. Association mismatch:- Table relationship problem

Granularity mismatch: HAS-A relationship problem:

At the Object level, we have the Has-A relationship but at the table level, there is no Has-A relationship. This mismatch is also known as Granularity Mismatch.

Example:

```
//Fine Grain class //Address.java public class Address{ //This is a value object, not an Entity private String city; private String state; private String pincode; //getter and setters } //Coarse grain class //Employee.java @Entity public class Employee{ //this is an Entity class @Id private int eid; private String ename; private int salary private Address addr; // has-A relationship }
```

Solution for the above Has-A relationship problem:

Approach1:

- We need to create a single table with all the columns (all for coarse grain + all for fine grain classes)
- Apply **@Embeddable** at the top of the Address class or **@Embedded** at the top of the **Address addr** variable inside the Employee Entity.

Example:

```
@Entity public class Employee { @Id @GeneratedValue(strategy=GenerationType.AUTO)
private int eid; private String ename; private int salary; @Embedded private Address
addr; //getters and setters }
```

EMUtil.java:

```
public class EMUtil { private static EntityManagerFactory emf; static{
emf=Persistence.createEntityManagerFactory("emp-unit"); } public static EntityManager
provideEntityManager(){ //EntityManager em= emf.createEntityManager(); //return em;
return emf.createEntityManager(); } }
```

Demo.java:

```
public class Demo { public static void main(String[] args) { EntityManager em=
EMUtil.provideEntityManager(); Employee emp=new Employee(); emp.setEname("Ram");
emp.setSalary(7800); emp.setAddr(new Address("Maharastra", "pune", "75455")); //Address
adr=new Address("maharastra", "pune","75455"); //emp.setAddr(adr);
em.getTransaction().begin(); em.persist(emp); em.getTransaction().commit();
System.out.println("done..."); } }
```

- If we try to take 2 addresses (one for home and another for office) and then try to persist the employee object then we will get an exception "**repeated column**"
- We can solve this problem by overriding the column names of Embedded object by using the **@AttributeOverrides** annotation.

Example:

Employee.java:

```
@Entity public class Employee { @Id @GeneratedValue(strategy=GenerationType.AUTO)
private int eid; private String ename; private int salary; @Embedded
@AttributeOverrides({
@AttributeOverride(name="state",column=@Column(name="HOME_STATE")),
@AttributeOverride(name="city",column=@Column(name="HOME_CITY")),
@AttributeOverride(name="pincode",column=@Column(name="HOME_PINCODE")) }) private
Address homeAddr; @Embedded @AttributeOverrides({
@AttributeOverride(name="state",column=@Column(name="OFFICE_STATE")),
@AttributeOverride(name="city",column=@Column(name="OFFICE_CITY")),
@AttributeOverride(name="pincode",column=@Column(name="OFFICE_PINCODE")) }) private
Address officeAddr; // getters and setters }
```

Demo.java:

```
public class Demo { public static void main(String[] args) { EntityManager em=
EMUtil.provideEntityManager(); Employee emp=new Employee(); emp.setEname("Ram");
emp.setSalary(7800); emp.setHomeAddr(new Address("Maharastra", "pune", "75455"));
emp.setOfficeAddr(new Address("Telengana","hydrabad", "785422"));
em.getTransaction().begin(); em.persist(emp); em.getTransaction().commit();
System.out.println("done..."); } }
```

Approach2:

If any employee has more than two addresses then taking too many columns inside a table will violate the rules of normalization.

To solve this problem we need to use the **@ElementCollection** annotation, and let the user add multiple addresses using List or Set.

In this case, ORM s/w will generate a separate table to maintain all the address details with a foreign key that refers the primary key of Employee table.

Example:

Employee.java

```
@Entity public class Employee { @Id @GeneratedValue(strategy=GenerationType.AUTO)
private int eid; private String ename; private int salary; @ElementCollection @Embedded
private Set<Address> addresses=new HashSet<Address>(); // getters and setters }
```

Note: It is recommended to override the equals() and hashCode() methods if we want to put any user-defined objects inside the HashSet or a key of the HashMap.

Address.java:

```
package com.masai.model; import java.util.Objects; public class Address { private String state; private String city; private String pincode; private String type; @Override public int hashCode() { return Objects.hash(city, pincode, state, type); } @Override public boolean equals(Object obj) { if (this == obj) return true; if (obj == null) return false; if (getClass() != obj.getClass()) return false; Address other = (Address) obj; return Objects.equals(city, other.city) && Objects.equals(pincode, other.pincode) && Objects.equals(state, other.state) && Objects.equals(type, other.type); } public String getState() { return state; } public void setState(String state) { this.state = state; } public String getCity() { return city; } public void setCity(String city) { this.city = city; } public String getPincode() { return pincode; } public void setPincode(String pincode) { this.pincode = pincode; } public String getType() { return type; } public void setType(String type) { this.type = type; } public Address(String state, String city, String pincode, String type) { super(); this.state = state; this.city = city; this.pincode = pincode; this.type = type; } public Address() { // TODO Auto-generated constructor stub } @Override public String toString() { return "Address [state=" + state + ", city=" + city + ", pincode=" + pincode + ", type=" + type + "]"; } }
```

Demo.java:

```
public class Demo { public static void main(String[] args) { EntityManager em= EMUtil.provideEntityManager(); Employee emp=new Employee(); emp.setEname("Ram"); emp.setSalary(7800); Employee emp= new Employee(); emp.setEname("Ramesh"); emp.setSalary(6800); emp.getAddresses().add(new Address("Mh", "Pune", "787887","home")); emp.getAddresses().add(new Address("MP", "Indore", "584542","office")); em.getTransaction().begin(); em.persist(emp); em.getTransaction().commit(); System.out.println("done..."); } }
```

When we execute the above application, 2 tables will be created:

1. **employee** table: which will contain only Employee details (it will not contain any details of any address)
2. **employee_addresses** table: this table will contain the details of all the addresses with a foreign column **employee_eid** which refer to the **eid** column of the **employee** table.

Note: If we want to change the 2nd table **employee_addresses** and the foreign key column with our choice name then we need to use **@JoinTable** and **@JoinColumn** annotations.

Example:

```
@Entity public class Employee { @Id @GeneratedValue(strategy=GenerationType.AUTO)
private int eid; private String ename; private int salary; @ElementCollection @Embedded
@JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp_id")) private
Set<Address> addresses=new HashSet<Address>(); //getters and setters }
```

With the above example, the 2nd table will be created by the name **empaddess** and the foreign key column will be by the name **emp_id**.

Example: Getting all the Addresses of an Employee whose name is Ramesh.

```
public class Demo { public static void main(String[] args) { EntityManager em=
EMUtil.provideEntityManager(); //get all the Address of a Employee whose name is Ramesh
String jpql="from Employee where ename='Ramesh'"; Query q= em.createQuery(jpql);
List<Employee> emps= q.getResultList(); for(Employee emp:emps) { Set<Address> addrs=
emp.getAddresses(); for(Address adr:addrs) { System.out.println(adr); } } em.close(); }
}
```

Eager and Lazy loading:

By default ORM s/w (Hibernate) perform lazy loading while fetching the objects, when we fetch the parent object(first-level object), then only the first-level object-related data will be loaded into the memory, but the 2nd level object-related data will be loaded at the time of calling the 2nd level object related methods.

Example:

```
public class Demo { public static void main(String[] args) { EntityManager em=
EMUtil.provideEntityManager(); Employee emp= em.find(Employee.class, 10); em.close(); //
even though before closing the EM obj we got the Employee obj //here only Employee
related obj will be loaded, address obj data will not be loaded //so while fetching the
address-related data we will get an exception. System.out.println(emp.getEid());
System.out.println(emp.getEname()); System.out.println(emp.getSalary());
System.out.println("All Address are:-");
System.out.println("============================"); Set<Address> addreses=
emp.getAddresses(); for(Address ad:addreses){ System.out.println("city :"+ad.getCity());
System.out.println("state :"+ad.getState()); System.out.println("Pincode
:"+ad.getPincode()); System.out.println("****************************"); }
System.out.println("done..."); } }
```

Note: internally it loads all the first-level object and provides a proxy object for the second-level object.

To solve the above problem we need to use Eager loading:

```
Employee.java:- ------------------ @Entity public class Employee { @Id
@GeneratedValue(strategy=GenerationType.AUTO) private int eid; private String ename;
private int salary; @ElementCollection(fetch=FetchType.EAGER) @Embedded
@JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp_id")) private
Set<Address> addresses=new HashSet<Address>(); //getters and setters }
```

Inheritance mismatch :- IS-A relationship problem

At Object level we have the IS-A relationship possible exist but at table level we don't have the IS-A relationship.

To address this, the JPA specification provides several strategies:

- **Single Table** – The entities from different classes with a common ancestor are placed in a single table.
- **Joined Table** – Each class has its table, and querying a subclass entity requires joining the tables.
- **Table per Class** – All the properties of a class are in its table, so no join is required.
- ***MappedSuperclass*** – the parent classes, can't be entities

Each strategy results in a different database structure.

Entity inheritance means that we can use polymorphic queries for retrieving all the subclass entities when querying for a superclass.

Single Table:

The Single Table strategy creates one table for each class hierarchy. JPA also chooses this strategy by default if we don't specify one explicitly.

We can define the strategy we want to use by adding the *@Inheritance* annotation to the superclass:

Example:

```
//MyProduct.java //parent entity @Entity @Inheritance(strategy =
InheritanceType.SINGLE_TABLE) public class MyProduct { @Id private long productId;
private String name; // constructor, getters, setters }
```

Then we can add the subclass entities:

```
//Book.java @Entity public class Book extends MyProduct { private String author; }
//Pen.java @Entity public class Pen extends MyProduct { private String color; }
```

Discriminator Values:

Since the records for all entities will be in the same table, **Hibernate needs a way to differentiate between them.**

By default, this is done through a discriminator column called *DTYPE* that has the name of the entity as a value.

To customize the discriminator column, we can use the *@DiscriminatorColumn* annotation:

```
@Entity(name="products") @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="product_type", discriminatorType = DiscriminatorType.INTEGER)
public class MyProduct { // ... }
```

Here we've chosen to differentiate *MyProduct* subclass entities by an *integer* column called *product_type*.

Next, we need to tell Hibernate what value each subclass record will have for the *product_type* column:

```
@Entity @DiscriminatorValue("1") public class Book extends MyProduct { // ... } Copy
@Entity @DiscriminatorValue("2") public class Pen extends MyProduct { // ... }
```

Joined Table:

Using this strategy, each class in the hierarchy is mapped to its table. The only column that repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

Let's create a superclass that uses this strategy:

```
@Entity @Inheritance(strategy = InheritanceType.JOINED) public class Animal { @Id
private long animalId; private String species; // constructor, getters, setters }
```

Then we can simply define a subclass:

```
@Entity public class Pet extends Animal { private String name; // constructor, getters,
setters }
```

Both tables will have an *animalId* identifier column.

The primary key of the *Pet* entity also has a foreign key constraint to the primary key of its parent entity.

To customize this column, we can add the *@PrimaryKeyJoinColumn* annotation:

```
@Entity @PrimaryKeyJoinColumn(name = "petId") public class Pet extends Animal { // ... }
```

The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables, which can result in lower performance for large numbers of records.

The number of joins is higher when querying the parent class because it will join with every single related child — so performance is more likely to be affected the higher up the hierarchy we want to retrieve records.

Table per Class:

The Table per Class strategy maps each entity to its table, which contains all the properties of the entity, including the ones inherited.

The resulting schema is similar to the one using @MappedSuperclass. But Table per Class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result.

To use this strategy, we only need to add the *@Inheritance* annotation to the base class:

```
@Entity @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) public class Vehicle {
@Id private long vehicleId; private String manufacturer; // standard constructor,
getters, setters }
```

Then we can create the subclasses in the standard way.

This is not that different from merely mapping each entity without inheritance. The distinction is clear when querying the base class, which will return all the subclass records as well by using a *UNION* statement in the background.

The use of *UNION* can also lead to inferior performance when choosing this strategy. Another issue is that we can no longer use identity key generation.

MappedSuperclass:

Using the *MappedSuperclass* strategy, inheritance is only evident in the class but not the entity model.

Let's start by creating a *Person* class that will represent a parent class:

```
@MappedSuperclass public class Person { @Id private long personId; private String name;
// constructor, getters, setters }
```

Notice that this class no longer has an *@Entity* annotation, as it won't be persisted in the database by itself.

Next, let's add an *Employee* subclass:

```
@Entity public class MyEmployee extends Person { private String company; // constructor,
getters, setters }
```

In the database, this will correspond to one *MyEmployee* table with three columns for the declared