Advanced Hibernate

It deals with applying object oriented concepts

- Inheritance
- Association

to Database.

Popularly known as **Solving Impedance Mismatch Issue.**

JPA (Hibernate) Inheritance Strategies

Inheritance is one of the fundamental design principles in OOP. BUT relational databases don't support inheritance.

JPA suggests different strategies to support inheritance hierarchies.

1 **Single Table Strategy** (**default inheritance strategy**)

- In this inheritance, a single table is used to store all the instances of the entire inheritance hierarchy.
- The Table will have a column for every attribute of every class in the hierarchy.
- Discriminator columns identify which class a particular row belongs.

Annoations used in **super class**

```
@Entity
@Inheritance// Default strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "emp_type")
@Table(name = "employees")
public class Employee {....}
```

In sub class :

```
@Entity
@DiscriminatorValue("worker")
public class Worker extends Employee {..}

@Entity
@DiscriminatorValue("mgr")
public class Manager extends Employee {..}
```

With this mapping strategy

- Only a single table will be created for both concrete classes (Manager and Worker).
- Hibernate will create a discriminator column named emp_type to differentiate each concrete type.
- The value of this column will be either worker or mgr

Use case

- In case of larger number of common fields between sub classes.

2. **Joined Table Strategy**

- It mirrors the object structure in the database.
- In this approach, a separate database table is defined for each of the class in the hierarchy
- Each table stores only its local attributes.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Product {
   @Id @GeneratedValue
   private Long id;
   private String name;
}

@Entity
public class Book extends Product {
   private String isbn;
}
```

- The above mapping will create two tables
- One for super class Product and another for entity Book.
- Both the tables will have Product id column.
- The primary key of table Book will have foreign key relationship with primary key of Product table.

3. **Table Per class**

- Not supported by all JPA vendors .
- A separate table is defined for each concrete class in the inheritance hierarchy to store all the attribute of that class and all its super classes.

4. **@MappedSuperClass**

- Extract common behaviour in a super class (can be concrete or abstract)
- A mapped super class has no separate table defined for it.
- All other entities can extend from this super class.

Eg.

```
@MappedSuperclass
public class Person {
   @Id
   private Long id;
   private String name;
}

@Entity
public class Employee extends Person {
   private String company; }
```

- The above configuration will result in a single table for employee with 3 columns (id, name and company)
- Person class will never have its own table in this case.

Reference Case Study HealthCare

Development Steps

1. **Apply Inheritance**
   - Create BaseEntity, as common abstract | concrete super class
   - All other entities can extend from it.
   - No separate table required for it.
   - Add common fields.
   - Eg.

   - ID field
   @Id @GeneratedValue (strategy=GenerationType.IDENTITY)

   - creation time stamp (To maintain date | time | timestamp of when was the entity inserted in DB)

   @CreationTimestamp

   - updation timestamp

   @UpdateTimestamp

   - version field (used in optimistic locking)

   @Version

2. **Association between Entities**

   - You can establish uni directional as well as bi directional association between Entities
   - Easier configuration will be for the uni directional association

One to One unidirectional association between Entities

Project's data navigation requirements will decide the direction.

Eg. In the Healthcare domain

- You will need to access user details from Patient | Doctor | Admin more often than from User to other entities.
- So it is better to navigate -
- Patient HAS-A User Patient 1---->1 User
- Doctor HAS-A User (Doctor 1-----> 1 User)

In one-one association

- You can configure **ANY** side as the owner of the association.
- No explicit rule saying which side should be the owner.

What the **owning side** ?

- The side containing the physical mapping (FK) .

Code Sample

In Patient class

- extends BaseEntity
- patient specific fields
- Add association field

  private User userDetails;

1. **Problem** - After adding associations & starting Hibernate SessionFactory

- Hibernate throws – org.hibernate.MappingException

Cause

- Hibernate can't figure out the type of association (i.e whether it is one-one or many-one)

Solution

- Add Mapping annotations
- @OneToOne

2. By default, the name of Foreign key is decided by hibernate

To customize Foreign key name & add NOT NULL constraint

- @JoinColumn(name=”FKColName: , nullable=false)

Final example in Patient class ,
@OneToOne
@JoinColumn(name=”user_id”,nullable=false)
private User userDetails;

3. Use Hibernate property , hbm2ddl.auto=create , in development phase

- Hibernate drops existing tables & creates new tables , upon application restart (i.e while creating SF)
- Use it in development phase only

4. In a similar , establish unidirectional association between

Doctor 1---->1 User

Final example in Doctor class ,
@OneToOne
@JoinColumn(name=”user_id”,nullable=false)
private User userDetails;

5. Appointment Entity

- extends BaseEntity

- add its specific fields (eg. status, time slot)

- Add associations (Many appointments can be taken by 1 Patient. Many Appointments can be issued by 1 Doctor)

5.1 **Uni dir association from**

- Appointment * ------> 1 Doctor (many to one)

```
@ManyToOne
@JoinColumn(name="doctor_id",nullable = false)
 private Doctor myDoctor;
```

5.2 Uni dir association from Appointment * ------> 1 Patient (many to one)

- Appointment * ------> 1 Patient (many to one)

```
@ManyToOne
@JoinColumn(name="patient_id",nullable = false)
private Patient myPatient;
```

NOTE

Hibernate developer has to simply identify the associations , add suitable annotations . Actual table creations , PK , FK , join tables , not null constraints will be automatically added by Hibernate

- Classic example of abstraction

- This is how Hibernate solves the issue of impedance mismatch

6. Many to many , unidirectional association between

Appointment *------->*   DiagTest

- Hibernate throws MappingException without any annotations , since it is unable to figure out is it the association between entity or non entity & doesn't know one-many or many-many type.

Solution

Add mapping annotation , in Appointment class

```
@ManyToMany
private Set<DiagTest> diagTests=new HashSet<>();
```

- Set is a preferred type of the Collection , in many-many association over List
- Reduces no of select queries , especially while removing the element .
- Add equals & hashCode in DiagTest class.

  - Using Lombok annotation

9. To customize name of the join table & composite PK column names , @ManyToMany

```
@JoinTable(name="appointment_tests",joinColumns=@JoinColumn(name="appointment_id"),inverseJoinColumns=@JoinColumn(name="test_id"))

private Set<DiagTest> diagTests=new HashSet<>();
```

10. In any of the entity sub classes , to optionally modify inherited column name

- Eg. Doctor extends BaseEntity. To override its inherited PK column name from id to “doctor_id”
- Use
  - **@AttributeOverride(name = "id", column = @Column(name = "doctor_id"))**

11. **Cascading**

- It refers to the ability to automatically propagate the state of an entity across associations between entities.

- It is a powerful feature that simplifies the management of complex E-R within a persistence context, reducing boiler plate code

- It allows you to automatically propagate certain operations (like saving, updating, and deleting) to associated entities, reducing the boilerplate code .

- Example Case

  Patient HAS –A User (Patient -> User)
  If you directly save Patient details in the DAO layer without explicitly saving User Details first, Hibernate throws
  - org.hibernate.TransientPropertyValueException

  Meaning , you will have to explicitly save User details & then Patient details.
  Instead, specify Cascading.

  When you define a relationship between two entitiesand specify a cascade type, Hibernate will automatically perform the specified operation on the related entities when you perform that operation on the source entity.

Types of cascading options via jakarta.persistence.CascadeType enum. Constants

- ALL,PERSIST, MERGE, REMOVE, REFRESH, DETACH

CascadeType.PERSIST:
Use case - When you want to persist a new parent entity and its associated child entities in a single transaction.
CascadeType.MERGE:
Use Case: When you want to update an existing parent entity and its associated child entities in a single transaction.
CascadeType.REMOVE:
Use Case: When you want to delete a parent entity and its associated child entities in a single transaction.
CascadeType.REFRESH:
Use Case: When you want to refresh the state of a parent entity and its associated child entities from the database.
CascadeType.ALL:
Use Case: When you want to apply all of the above cascade types to the

Revisited example in Patient class ,
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name=”user_id”,nullable=false)
private User userDetails;

12. **JPA (Hibernate) bidirectional relationship one to many**

- A JPA bidirectional one-to-many relationship allows navigation and access to associated entities from **both sides** of the relationship.
- Eg. From Doctor to Appointment and from Appointment back to Doctor
- It is achieved using the @OneToMany annotation on the "one" side and the @ManyToOne annotation on the "many" side, with one side designated as the **owning side**.

Key Concepts

- **Bidirectional:** Both entity classes have a field that references the other.
- **Owning Side:** The side that is responsible for managing the relationship's foreign key in the database table. In a standard bidirectional one-to-many mapping, the @ManyToOne side is always the owning side.
- **Inverse Side:** The @OneToMany side is typically the inverse side and uses the **mappedBy** attribute to indicate which field in the owning entity manages the relationship.
- If the **mappedBy** attribute is not specified, Hibernate created extra Mapping Table , containing both of the Primary keys.

Example case

Consider a Doctor entity and an Appointment entity. One doctor can give many appointments, but 1 appointment belongs to one doctor only.

Doctor Entity (the "one" side, inverse side)

This entity holds a collection of appointments and uses mappedBy to specify it's not the owner of the relationship's foreign key.

Eg. In Doctor class

```
   @OneToMany(mappedBy = "myDoctor", cascade = CascadeType.ALL, orphanRemoval = true)
   private List<Appointment> appointments = new ArrayList<>();
```

It is also recommended to add helper methods , to add or remove appointment entity.

Appointment Entity (the "many" side, owning side)

This entity has a single reference back to its Doctor and contains the @JoinColumn annotation to specify the foreign key column name in the appointments table.

```
   @ManyToOne
   @JoinColumn(name = "doctor_id")
   private Doctor myDoctor;
```

What is **orphanRemoval** ?

A boolean property of @OneToMany / @OneToOne annotation.
default value - false
It specifies
Whether to apply the remove operation to entities that have been removed from the relationship and to cascade the remove operation to those entities.

13. **org.hibernate.LazyInitializationException**

Example case

Consider above bidirectional association between Doctor & Appointment

- Doctor 1<---→* Appointment

Objective

- Display doctor details & list of his/her upcoming
  - Input - doctor id

Problem

In above case , Hibernate throws
- LazyInitializationException , while accessing - appointment details

Cause

- JPA (Hibernate) supports default data fetching policies
one -one : EAGER
one-many : LAZY

many-one : EAGER
many-many : LAZY

In this case one -> many (LAZY)

- When a select query is fired on the doctors table , due to session.find(Doctor.class,doctorId)
- Hibernate DOES NOT automatically fetch the data from appointments table.
- For optimal performance.
- Hibernate creates a dynamic proxy using helper byte-buddy jar, to represent un fetched data from DB.
- Any time , you are accessing un fetched data (Proxy) , outside the session scope , Hibernate throws LazyInitializationException.

Solutions
1. **Entity layer solution**
- Change fetching policy from LAZY->EAGER
- Add fetch=FetchType.EAGER , in @OneToMany annotation.
Use case -
When the size of many is small.

2. **DAO layer solution**
- Simply Access size of the collection, within session scope (i.e before commit)

Disadvantage -
Hibernate gets complete data (doctor + appointments) in multiple queries.

3. Best solution is
- Use "join fetch" in JPQL
- Eg.
jpql- select d from Doctor d left join fetch d.appointments where d.id=:docId
Hibernate will fetch doctor details & associated appointment details in a single join query.