Inheritance in JPA

Till now we were looking at composition or aggregation way of defining the relationship.

We will see example of Resources(id, name, size, url) as parent and it children as Video(length), File(type), and Text(content)

We can use multiple strategy for that: single table, join table, table per class.

Pros:
1. Refuse of common code among all the entities
2. Simplified queries spanning multiple entities

Cons:
1. Make the database schema more complex especially when we are using join table per class or table per class strategy
2. Second is rigidity (difficult to change base or child class because it can affect the inheritance hierarchy)

Composition is more flexible and easier to change than inheritance as it involves creating a class that has reference to one or more subjects and delegate tasks to these subjects. This allows to combine the functionality of multiple classes into a single class without inheritance hierarchy so we can change the object that a class delegates without affecting the class itself.

Let say we make a base entity BaseEntity which contain common fields like id, createdAt, lastModifiedAt, lastModifiedBy, createdBy.
Now we will annotate this with @MappedSuperClass which indicates that the class is super class that is not mapped to a database table.
So this entity exists only in the code and not in database for reusability. It cannot be queries or persisted directly.

Now to make use of builder pattern in children so that they can fill up the properties in the super class we have to use @SuperBuilder for all the entities that are of concerned so that child builder class can access all the properties.

When we use inheritance and @Data (Lombok annotation) we have to include @EqualsHashCode(callSuper = true)

Single Table Strategy: map an inheritance hierarchy to entities to a single database table. All subclasses of the inheritance hierarchy are mapped to the same table and a discriminator column is used to distinguish between different subclasses. All the properties of the super class and sub class will be persisted to a single table. It leads to inefficient query and large table size. It is suitable

for when inheritance hierarchy is not deep.
Add @Inheritance(strategy = InheritanceType.SINGLE_TABLE)

To make up say Text class we have to fill up the properties of both Text and Resource classes.
Now there is need for discriminator value.

By default discriminator column name for this will be dtype.
We can modify this by specifying annotation over Base class like Resource @DiscriminatorColumn(name = “resource_type”)
Now to give discriminator value -> e.g. @DescrimintorValue(“V”) for video, @DescriminatorValue(“F”) for file, etc. should be specified over respective subclass.

So when we create Video entity and save it, jpa will automatically fill the resource_type with “V”.

Join Table Strategy: Each sub class is mapped to different table with foreign key pointing to base table. It makes the queries efficient but gives us more table. Useful when subclasses has remarkable difference and they are many in number.

Just change require, remove discriminator thing and make @Inheritance(strategy = InheritanceType.JOINED)
The foreign key also acts as the primary key for the table.
Jpa automatically inserts into resource if we insert only video.

To change the name of foreign key (by default only id) we can do using @PrimaryKeyJoinColumn(name = “video_id”) over sub class.

Table per class: Each concrete subclass is mapped to a separate table. Abstract classes are not mapped to separate table and their properties are inherited by concrete classes. Each concrete subclass table will have its specific properties along with the properties of the super class.
It is used to optimise the performance of the query when we have small number of subclasses with major differences. So only 1 insert will take place as per the instance type. We have to change the type of strategy to InheritanceType.TABLE_PER_CLASS and remove PrimaryKeyColumn annotation as it works only for Joined strategy. In this we will not have any foreign key.

Query:
But now if we fetch the base class we will also get all the sub classes which sometimes we do not want and make the query slow.
Over the sub class we can specify @Polymorphism(type = PolymorphismType.EXPLICIT)

/**

In JPA, performing a polymorphic query allows you to retrieve instances of both the superclass and its subclasses using a single query. This is useful when you have an inheritance hierarchy and want to retrieve all entities that belong to the hierarchy regardless of their specific type.

Here's how you can perform a polymorphic query in JPA:

1. **Using JPQL (Java Persistence Query Language)**:
   - JPQL supports polymorphic queries through the `TYPE` operator, which allows you to query for entities of a specific type or its subclasses.
   - Here's an example of a polymorphic query in JPQL:
     ```java
     String jpql = "SELECT e FROM Employee e WHERE TYPE(e) IN (:types)";
     List<Employee> employees = entityManager.createQuery(jpql,
Employee.class)
                                     .setParameter("types", Arrays.asList(Employee.class,
Manager.class))
                                     .getResultList();
     ```
     In this example, `Employee` is the superclass, and `Manager` is a subclass. The query retrieves all instances of `Employee` and its subclasses.

2. **Using Criteria API**:
   - The Criteria API also supports polymorphic queries through the `type()` method, which allows you to specify the type of entity you want to query for.
   - Here's an example of a polymorphic query using Criteria API:
     ```java
     CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
     CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
     Root<Employee> root = criteriaQuery.from(Employee.class);
     criteriaQuery.select(root);

criteriaQuery.where(criteriaBuilder.in(root.type()).value(Employee.class).value(
Manager.class));
     List<Employee> employees =
entityManager.createQuery(criteriaQuery).getResultList();
     ```
     This Criteria API query retrieves all instances of `Employee` and its subclasses.

3. **Using Native SQL**:
   - You can also perform polymorphic queries using native SQL queries by joining the tables corresponding to the superclass and its subclasses.

- However, this approach is less portable across different database systems and requires knowledge of the underlying database schema.

Polymorphic queries allow you to retrieve entities from an inheritance hierarchy without needing to know their specific types in advance, providing flexibility and ease of use when working with complex object models.

When you have an abstract class in an inheritance hierarchy and you want to perform a polymorphic query to retrieve instances of both the abstract class and its subclasses, you can still use JPQL (Java Persistence Query Language) or Criteria API. However, you need to be mindful of a few considerations:

1. **Query for Subclasses**: Since instances of the abstract class itself cannot exist in the database, you'll need to query for instances of its concrete subclasses.

2. **Use Polymorphic Queries**: Utilize the `TYPE` operator in JPQL or the `type()` method in Criteria API to specify the type of entities you want to retrieve.

Here's how you can perform a polymorphic query when there is an abstract class in the hierarchy:

1. **Using JPQL**:
   ```java
   String jpql = "SELECT e FROM BaseEntity e WHERE TYPE(e) IN (:types)";
   List<BaseEntity> entities = entityManager.createQuery(jpql, BaseEntity.class)
                                    .setParameter("types",
Arrays.asList(ConcreteEntity1.class, ConcreteEntity2.class))
                                    .getResultList();
   ```
   Replace `BaseEntity` with the name of your abstract class and `ConcreteEntity1`, `ConcreteEntity2`, etc., with the names of your concrete subclasses.

2. **Using Criteria API**:
   ```java
   CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
   CriteriaQuery<BaseEntity> criteriaQuery =
criteriaBuilder.createQuery(BaseEntity.class);
   Root<BaseEntity> root = criteriaQuery.from(BaseEntity.class);
   criteriaQuery.select(root);
   criteriaQuery.where(criteriaBuilder.in(root.type())
                               .value(ConcreteEntity1.class)
                               .value(ConcreteEntity2.class));
   List<BaseEntity> entities =
   ```

entityManager.createQuery(criteriaQuery).getResultList();
   ```
   Replace `BaseEntity` with the name of your abstract class and `ConcreteEntity1`, `ConcreteEntity2`, etc., with the names of your concrete subclasses.

By specifying the concrete subclasses in the `TYPE` or `type()` clause, you can retrieve instances of both the abstract class and its subclasses in a polymorphic query.

If you change the inheritance strategy from `JOINED` to `TABLE_PER_CLASS`, it will impact the way entities are mapped to database tables. Here's what happens:

1. **Table Structure**:
   - In the `TABLE_PER_CLASS` strategy, each concrete entity (including the abstract class) in the inheritance hierarchy is mapped to its own table in the database.
   - This means that there won't be a separate table for the abstract class; instead, each concrete subclass will have its own table containing all fields inherited from the abstract class along with its own specific fields.

2. **No Discriminator Column**:
   - Unlike the `JOINED` strategy, there is no need for a discriminator column in the `TABLE_PER_CLASS` strategy because each subclass has its own table.
   - In the `TABLE_PER_CLASS` strategy, the database schema doesn't need to store information about the concrete class type for each row since each table represents a single concrete class.

3. **Polymorphic Queries**:
   - Polymorphic queries may be less straightforward in the `TABLE_PER_CLASS` strategy because there's no shared table for all entities in the hierarchy.
   - You would need to query each table individually and then combine the results programmatically to achieve polymorphic behavior.
   - However, you can still query individual tables using their respective entity types to retrieve instances of specific subclasses.

4. **Performance Considerations**:
   - In the `TABLE_PER_CLASS` strategy, each concrete subclass has its own table, which can lead to a larger number of tables in the database.
   - This may impact performance, especially when querying across multiple tables or when dealing with large hierarchies with many subclasses.

5. **Table Design**:
   ```

   - With `TABLE_PER_CLASS`, each concrete subclass table will contain all fields inherited from the abstract class, resulting in denormalization.
   - This can lead to duplication of data and potential data integrity issues if not managed carefully.

In summary, changing the inheritance strategy to `TABLE_PER_CLASS` will result in each concrete subclass having its own table in the database, removing the need for a discriminator column but potentially leading to a larger number of tables and denormalization of the schema. Polymorphic queries may require querying individual tables and combining results programmatically.

When you have an inheritance hierarchy with an abstract class and you're using the inheritance strategy of `JOINED`, you cannot directly insert rows into the table corresponding to the abstract class because it's an abstract entity and doesn't have a table associated with it. Instead, you'll need to insert rows into the tables corresponding to the concrete subclasses.

Here's a general approach to insert rows when there's an abstract class in an inheritance hierarchy with `JOINED` strategy:

1. **Instantiate a Concrete Subclass**:
   - Choose one of the concrete subclasses of the abstract class.
   - Instantiate an object of this concrete subclass.

2. **Set Values**:
   - Set values for properties specific to the concrete subclass.
   - You can also set values for properties inherited from the abstract class if they have setters.

3. **Persist the Entity**:
   - Use JPA's `EntityManager` to persist the entity.

Here's an example:

Suppose you have an abstract class `Vehicle` and concrete subclasses `Car` and `Truck`. You want to insert a new `Car` entity into the database.

```java
// Abstract class
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // Other properties and methods
}

// Concrete subclass
@Entity
public class Car extends Vehicle {
    private String model;

    // Getters and setters
}

// Inserting a new Car entity
Car car = new Car();
car.setModel("Toyota Camry");

entityManager.persist(car); // Persist the Car entity
```

In this example, you're inserting a new `Car` entity. The `Car` class inherits from the `Vehicle` abstract class. You create a new instance of `Car`, set its properties (including those inherited from `Vehicle`), and persist it using the `EntityManager`.

Remember, since `Vehicle` is an abstract class and doesn't have its own table, you cannot directly insert rows into the `Vehicle` table. You must choose one of its concrete subclasses (`Car`, `Truck`, etc.) and insert rows into the respective tables for those concrete subclasses.

*/

https://www.baeldung.com/hibernate-inheritance