Introduction to Hibernate and Relationships

Hibernate Relationships

Hibernate allows developers to map Java objects to database tables and manipulate them using object-oriented programming concepts. One crucial aspect of developing Hibernate applications is dealing with relationships between objects and tables.

There are three types of relationships in Hibernate:

1. One To One Relationship
2. One To Many Relationships
3. Many To Many Relationships

One To One Relationship

In a one-to-one relationship, one entity is associated with one instance of another entity. An example of a source entity can be, at most, mapped to one instance of the target entity.

There can be two types of One To One Relationships:

1. Unidirectional Relationship

Unidirectional one-to-one Relationship is a type of Relationship where one entity is associated with exactly one instance of another. Still, while one entity is linked to another, the related entity does not necessarily have a corresponding link to the first entity.

For example, " Item " refers to a product or item being sold or managed within a system. In contrast, "ItemDetails" refer to additional information about that item, such as its name, description, price, and other details. The relationship between Item and Item Details is one-way, and Item Details only exist to provide more information about a specific Item.

Let's go through the code:

- The "@OneToOne" annotation specifies that there is a one-to-one relationship between the "Item" entity and the "ItemDetails" entity.

- The cascading attribute specifies that any changes made to the "Item" entity (e.g., deleting an "Item") will be cascaded to the associated "ItemDetails" entity so that it will be deleted as well.
- The getters and setters of "ItemDetails" are essential when we want to fetch and update “ItemDetails” associated with an "Item".

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {

    @OneToOne(cascade = CascadeType.ALL)
    private ItemDetails itemDetails;

    //Getters and setters of ItemDetails
}
```

2. Bidirectional Relationship

A Bidirectional one-to-one Relationship refers to a relationship between two entities where each entity has a reference to the other. Meaning there is an "OneToOne" relationship between two entities in which each entity can access the other. Considering the above example of an Item entity and an ItemDetails entity, we will need to implement a bidirectional one-to-one Relationship if we want to fetch details of the Item entity of a specific ItemDetail entity.

Let's review the code:

1. We must write additional code in ItemDetails Entity to link it back to Item Entity.
2. Mapped by specifies the bidirectional relationship between ItemDetails and Item Entities. Its value indicates the variable we want to link with from the Item entity.
3. We are using “CascadeType.ALL”, which means that any changes made to the "ItemDetails" entity will be cascaded to the associated "Item" entity.

```
//Item Details Entity
@Entity
@Table(name="item_details")
public class ItemDetails {

    @OneToOne(mappedBy ="itemDetails", cascade = CascadeType.ALL)
    private Item item;
}
```

One To Many Relationship

In a one-to-many relationship, one entity is associated with multiple instances of another entity. For example, consider an Item entity and an ItemReview entity. An Item may have multiple ItemReviews associated with it, and an ItemReview can only be associated with one Item. To map this relationship in Hibernate, we can add a **@OneToMany** annotation to the Item entity, specifying the ItemReview entity as the target of the relationship.

There can be two types of One To Many Relationships:

1. Unidirectional Relationship

In a unidirectional one-to-many relationship, a single entity on one side of the relationship is associated with multiple instances of another entity on the other side of the relationship, but the other entity does not have a direct reference back to the first entity.
For example, the Item entity can be associated with multiple instances of the ItemReview entity. However, the ItemReview entity does not have a direct reference back to the Item entity.

Let's go through the code:

- To implement this Relationship, we would typically add a List or Set the property to the Item entity to hold all the associated ItemReview entities.
- We will annotate this property with the @OneToMany annotation, which tells that this property represents a one-to-many relationship.
- The cascade attribute specifies that any changes made to the Item entity should also be applied to its associated ItemReview entities (e.g., if an Item is deleted, all its associated ItemReview entities should also be deleted).
- The JoinColumn annotation specifies the column in the ItemReview table.

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="item_id")
    private List<ItemReview> itemReview;

    //Getters and setters of ItemReview

}
```

2. Bidirectional Relationship

In a bidirectional one-to-many Relationship, one entity has a one-to-many relationship with another entity, and the other entity has a many-to-one relationship back to the first entity. Meaning that the "one" entity can have multiple related "many" entities, and each "many" entity can have a direct reference back to the associated "one" entity.

Considering the above example of an Item entity and an ItemReview entity, we must implement a bidirectional one-to-one Relationship to fetch the Item of the ItemReview entity of a specific ItemReview entity.

Let's go through the code:

- The mappedBy attribute in the @OneToMany annotation indicates that the ItemReview entity manages the relationship and will use the item property in the ItemReview entity as the foreign key to the Item entity.
- The cascade attribute specifies that any changes made to the Item entity should also be applied to its associated ItemReview entities (e.g., if an Item is deleted, all its associated ItemReview entities should also be deleted).

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {
    @OneToMany(mappedBy="item",cascade = CascadeType.ALL)
    @JsonManagedReference
    private List<ItemReview> itemReview;

    //Getters and setters of ItemReview
}
```

- The @ManyToOne annotation indicates that each ItemReview entity is associated with a single Item entity
- The @JoinColumn annotation specifies the column in the ItemReview table that contains the foreign key to the Item table.

```
//Item Details Entity
@Entity
@Table(name="item_review")
public class ItemReview {

    @ManyToOne
    @JoinColumn(name="item_id")
    @JsonBackReference
    private Item item;
}
```

Many To Many Relationship

In a many-to-many relationship, multiple instances of one entity are associated with multiple instances of another entity. For example, consider an Item entity and an Order entity. An Item may be associated with multiple Orders, and an Order may be associated with multiple Items. To map this relationship in Hibernate, we can add a @ManyToMany annotation to both the Item and Order entities, specifying the other entity as the target of the relationship. This will create a third table, a join table, that stores the associations between the two entities.

Let's review the code:

- The @ManyToMany annotation defines the relationship between Order and Item.
- The cascade attribute specifies the cascade behavior for any operations performed on the association. We don't want to delete any item when we delete an order so we won't use CascadeType.REMOVE, but deleting this order will delete the relationship between this order and all associated items from the order_item table.
- The relationship between these entities is established using a join table named order_item, which has foreign key columns referencing the primary keys of the Order and Item tables.
- The @JoinTable annotation specifies the name of the join table and the foreign key columns used to establish the relationship between the Order and Item entities. The joinColumns attribute specifies the column used to reference the Order table, while the inverseJoinColumns attribute specifies the column used to reference the Item table.

```
//OrderService
@Entity
@Table(name="orders")
public class Order {

	@ManyToMany(
	cascade={CascadeType.MERGE,CascadeType.PERSIST,CascadeType.REFRESH}
	)
	@JoinTable(name="order_item",
	joinColumns =@JoinColumn(name="order_id"),
	inverseJoinColumns = @JoinColumn(name="item_id"))
	private List<Item> items;
}
```

- The @ManyToMany annotation is used to define the relationship between Item and Order.
- The mappedBy attribute specifies that the relationship is mapped by the items property on the Order entity. This means that the join table and the foreign key columns are defined on the Order entity rather than the Item entity.

```
//Item Entity
@Entity
@Table(name="item")
public class Item {

        @ManyToMany(mappedBy = "items")
        @JsonIgnore
        private List<Order> orders;
}
```

- **Note 1:** The **@JsonIgnore** annotation is used to ignore the order's property when the Item entity is serialized to JSON, preventing a possible infinite circular reference between the Order and Item entities.
- **Note 2**: Before saving the order into the database, fetch all the item values and set those values to your new order.

```
//Order Service
@Transactional
public void saveOrder(Order order) {

    Order saveOrder =new Order();
    saveOrder.setOrderType(order.getOrderType());
    List<Item> itemList = new ArrayList<>();
    for(Item item:order.getItems()) {
       Item currentItem=itemDAL.getById(item.getId());
       itemList.add(currentItem);
    }
    saveOrder.setItems(itemList);
    orderDAL.save(saveOrder);
}
```

What is H2 Database?

H2 Database is an open-source, in-memory relational database management system written in Java. It is designed to be fast, lightweight, and easy to use, making it an ideal choice for testing and development.

In-memory databases store data in the computer's memory rather than on a disk. This means the database is created and stored in the computer's memory and lost when the program or application is closed. This makes it ideal for testing and development because it is easy to set up and use.

- The H2 database can be easily integrated with Java applications by adding the following dependency to the project's build file:
  After adding the dependency, you can configure the database connection properties in the application's configuration file, like the following example for a Spring Boot application:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- The configuration file sets up an H2 database with an in-memory storage model.
- The JDBC URL specifies that the H2 database is in memory and has the name ***"testdb"***.
- The driver class is a Java class that implements the JDBC API and provides connectivity to the database. In this case, we have specified the JDBC driver for the H2 database.
- Setting Hibernate dialect here for the H2 database is similar to setting the dialect for the MySQL database.
- The Hibernate ddl-auto property is set to ***"create"***, meaning the database schema will be created automatically when the application starts.
- The H2 console can be enabled by setting the ***Console.enabled*** property to true in the application configuration, this allows access to the H2 database console and interaction with the database during runtime.

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    spring.jpa.database-platform: org.hibernate.dialect.H2Dialect
    Hibernate.ddl-auto: create
```

Conclusion

In this lesson, we covered the basics of Hibernate and Hibernate. We discussed different types of relationships in Hibernate, including one-to-one, one-to-many, and many-to-many relationships. We also saw examples of unidirectional and bidirectional Relationships for these relationships. Furthermore, we discussed using the H2 database for testing and in-memory database operations and how to configure Hibernate with the H2 database.

Instructor Codes

- CNKart Application

References

1. Hibernate Relationships
2. H2 Database