Relationships

https://medium.com/@burakkocakeu/in-spring-data-jpa-onetomany-what-are-these-fields-mappedby-fetch-cascade-and-orphanremoval-2655f4027c4f

The `mappedBy` attribute is used in JPA entity mappings to establish bidirectional relationships between entities. It indicates that the entity on the other side of the relationship (the owning side) is responsible for managing the association.

When you have a bidirectional relationship between two entities, you typically define the relationship on one side as the owning side and the other side as the inverse side. The owning side is the one that contains the foreign key column in the database, while the inverse side does not.

Let's consider an example with two entities, `Parent` and `Child`, where `Parent` has a one-to-many relationship with `Child`:

```java
@Entity
public class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "parent")
    private List<Child> children;

    // other properties and methods
}

@Entity
public class Child {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private Parent parent;

    // other properties and methods
}
```

In this example:

- The `Parent` entity has a one-to-many relationship with `Child`, which means

that one parent can have multiple children.
- The `mappedBy` attribute in the `@OneToMany` annotation of the `Parent` entity indicates that the `Child` entity is the owning side of the relationship. It specifies the name of the field in the `Child` entity (`parent`) that owns the relationship.
- The `@ManyToOne` annotation in the `Child` entity establishes the many-to-one relationship with `Parent`. Since `Parent` is the owning side, there is no need to specify `mappedBy` in the `@ManyToOne` annotation.

By using `mappedBy`, you tell JPA that the `Parent` entity is not responsible for managing the association; instead, it's managed by the `parent` field in the `Child` entity. This helps avoid redundancy in the database schema and ensures consistency between the entities when managing bidirectional relationships.

In a many-to-many relationship between two entities in JPA, each entity can be associated with multiple instances of the other entity. This type of relationship typically requires a join table in the database to manage the association between the two entities.

When mapping a many-to-many relationship in JPA, you use the `@ManyToMany` annotation on both sides of the relationship. However, unlike one-to-many and many-to-one relationships, there's no concept of a "mappedBy" attribute in a many-to-many relationship because there's no owning or inverse side in the same way.

Let's illustrate with an example using two entities, `Student` and `Course`, where each student can enroll in multiple courses, and each course can have multiple students:

```java
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany
    @JoinTable(name = "student_course",
            joinColumns = @JoinColumn(name = "student_id"),
            inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses;

    // other properties and methods
}
```

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // other properties and methods
}
```

In this example:

- In the `Student` entity, the `@ManyToMany` annotation establishes the many-to-many relationship with the `Course` entity. The `@JoinTable` annotation specifies the name of the join table (`student_course`) and the columns used to join the tables.
- In the `Course` entity, the `@ManyToMany` annotation indicates the many-to-many relationship with the `Student` entity. The `mappedBy` attribute specifies the field in the `Student` entity (`courses`) that owns the relationship.

Notice that we use the `mappedBy` attribute in the `@ManyToMany` annotation in the `Course` entity to indicate that the `Student` entity is the owning side of the relationship. This is necessary because the `student_course` join table contains foreign keys to both the `Student` and `Course` tables, and we need to specify which side manages the association.

In summary, when mapping a many-to-many relationship in JPA, you use `@ManyToMany` on both sides of the relationship, and you specify the `mappedBy` attribute on one side to indicate the owning side of the relationship.

Sure, I'll provide examples of how to add entities in both one-to-many and many-to-one relationships.

1. **One-to-Many Relationship (Parent-Child)**:

Let's consider entities `Parent` and `Child` where `Parent` has a one-to-many relationship with `Child`.

```java
```

```
@Entity
public class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "parent", cascade = CascadeType.ALL)
    private List<Child> children = new ArrayList<>();

    // Getter and setter for children
}

@Entity
public class Child {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private Parent parent;

    // Getter and setter for parent
}
```

To add entities to the database:

```java
Parent parent = new Parent();
Child child1 = new Child();
Child child2 = new Child();

// Associate child entities with parent
child1.setParent(parent);
child2.setParent(parent);

// Add child entities to parent's collection
parent.getChildren().add(child1);
parent.getChildren().add(child2);

// Save parent entity along with associated child entities
entityManager.persist(parent);
```

2. **Many-to-One Relationship (Child-Parent)**:

Let's consider entities `Child` and `Parent` where `Child` has a many-to-one

relationship with `Parent`.

```java
@Entity
public class Child {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private Parent parent;

    // Getter and setter for parent
}

@Entity
public class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other properties

    // Getter and setter for properties
}
```

To add entities to the database:

```java
Parent parent = new Parent();
entityManager.persist(parent);

Child child1 = new Child();
child1.setParent(parent);
entityManager.persist(child1);

Child child2 = new Child();
child2.setParent(parent);
entityManager.persist(child2);
```

In both cases, when you persist the parent entity, associated child entities will also be persisted if you've properly set up cascading (`CascadeType.ALL` or specific cascading types) on the parent's collection in the one-to-many relationship, or on the many-to-one relationship if applicable. This ensures that the child entities are saved along with the parent entity in a single operation.

This mapping annotation signifies a one-to-one association between two entities, where one entity acts as the owner of the relationship, and the other entity is the inverse side. Let's break down this annotation:

```java
@OneToOne(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
```

- `@OneToOne`: Indicates a one-to-one association between entities.
- `mappedBy = "post"`: Specifies the name of the field in the inverse side entity (`mappedBy` attribute) that owns the relationship. This indicates that the `Post` entity is the owner of the relationship.
- `cascade = CascadeType.ALL`: Specifies that all entity state change operations (persist, merge, remove, refresh) performed on the owner side (`Post` entity) should be cascaded to the associated entity on the inverse side.
- `orphanRemoval = true`: Specifies that if the association from the owning entity (`Post`) to the associated entity is removed, the associated entity should be removed as well. In other words, if a `Post` entity is removed, the associated entity will also be removed from the database.

Here's an example to illustrate this mapping:

```java
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
    private PostDetails postDetails;

    // Other properties and methods
}

@Entity
public class PostDetails {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    private Post post;

    // Other properties and methods
}
```

In this example:

- The `Post` entity owns the one-to-one relationship, and it has a field `postDetails` of type `PostDetails` annotated with `@OneToOne`.
- The `mappedBy = "post"` attribute in the `@OneToOne` annotation of `Post` indicates that the `PostDetails` entity is the inverse side of the relationship, and the association is managed by the `post` field in `PostDetails`.
- `cascade = CascadeType.ALL` specifies that any state change operations performed on `Post` (such as persist, merge, remove) should be cascaded to `PostDetails`.
- `orphanRemoval = true` indicates that if a `Post` entity is removed, its associated `PostDetails` entity should also be removed from the database.

@MapIds

The `@MapsId` annotation in JPA is used to establish a mapping between the primary key of an entity and the primary key of another entity. This annotation is typically used in scenarios where one entity has a one-to-one relationship with another entity, and both entities share the same primary key value.

Here's a brief explanation of how `@MapsId` works:

1. **Shared Primary Key**: In a one-to-one relationship between two entities, you may want both entities to share the same primary key value. This is often the case when one entity represents a subset or an extension of another entity.

2. **Mapping Relationship**: The `@MapsId` annotation is applied to a relationship attribute in the dependent entity (the one with the foreign key referencing the primary key of the other entity). It specifies that the primary key value of the dependent entity should be mapped to the primary key of the referenced entity.

3. **Attribute Level**: The `@MapsId` annotation is applied at the attribute level and is typically used in conjunction with a `@JoinColumn` annotation to specify the foreign key column mapping.

4. **Usage Example**: Suppose you have entities `Employee` and

`EmployeeDetails`, where `EmployeeDetails` has a one-to-one relationship with `Employee` and shares the same primary key value. You can use `@MapsId` to map the primary key of `Employee` to `EmployeeDetails`.

Here's an example of how to use `@MapsId`:

```java
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "employee")
    private EmployeeDetails details;

    // Getters and setters
}

@Entity
public class EmployeeDetails {
    @Id
    private Long id; // This is the primary key, but it's also the foreign key referencing Employee

    @MapsId
    @OneToOne
    @JoinColumn(name = "id") // Foreign key column mapping
    private Employee employee;

    // Other attributes
    // Getters and setters
}
```

In this example:

- The `Employee` entity has a one-to-one relationship with `EmployeeDetails`.
- The `EmployeeDetails` entity has an attribute `employee` annotated with `@MapsId`, indicating that the primary key value of `EmployeeDetails` should be mapped to the primary key value of the referenced `Employee` entity.
- When you persist an `Employee` entity along with its associated `EmployeeDetails`, the primary key value of `Employee` will be automatically assigned to the `id` attribute of `EmployeeDetails`, establishing the one-to-one relationship with shared primary key values.