Theory: JPA Relationships

25 minutes 0 / 5 problems solved

§1. JPA Relationships

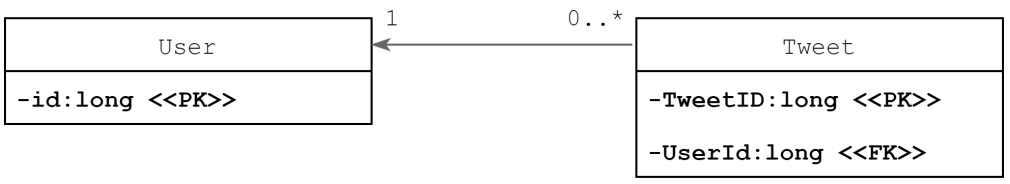
Up to this point, we've considered JPA capabilities for only one table in the database. However, in the real world, it is common to interact with several entities that are connected. There are four **types of connections** or relationships between two entities:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

JPA can manage each of these relationship types. For now, let's concentrate on **one-to-many** and **many-to-one** relationships.

§2. Unidirectional `@OneToMany`

Imagine you are developing Tweeter. A Tweeter user can post a lot of tweets, and each tweet can be posted by only one user. Such a relationship between the entities `User` and `Tweet` is called a **one-to-many** relationship: **one** user can post **many** tweets.



For defining the one-to-many relationship between the `User` and `Tweet` entities, you can use JPA `@OneToMany` annotation. This annotation is placed on top of the entity field or property that refers to the associated entity. Except for `@OneToMany` annotation, we need `@JoinColumn` annotation with the `"name = "` parameter for specifying a mapped column. It allows to join an entity association: such column connects `User` Table and `Tweet` table, so it is a `UserId` column in the `Tweet` table.

```
@Entity
public class User {

    @Id
    private long id;

    @OneToMany
    @JoinColumn(name = "UserID", nullable = false)
    private List<Tweet> tweets = new ArrayList<>();

}


@Entity
public class Tweet {


    @Id

    @Column(name = "TweetID")

    private long id;

}
```

Current topic:

JPA Relationships

Topic depends on:

✕ Introduction to JPA

Table of contents:

The `@JoinColumn` annotation on top of the tweets field indicates that the `Tweet` table has a `UserID` foreign key column specifying an entity association between these tables.

> Parameter `"nullable = false"` of the `@JoinColumn` annotation indicates that the annotated field shall not be null.

The `User` entity has a field `Tweets`. In other words, the `User` entity "knows" about `Tweets` to which it refers. However, the entity `Tweet` doesn't have any fields that would refer to the `User`, so `Tweet` doesn't "know" to which `User` it refers. Such a relationship is called **Unidirectional**. The unidirectional relationship has only an **owning side** meaning the side of the relation that *owns* the foreign key in the database.

§3. Bidirectional `@ManyToOne`

As you might have guessed, in the **Bidirectional** relationship, each entity "knows" about each other. It means that each entity has a field or property that refers to the other entity. The bidirectional relationship has both an owning side and an **inverse side**, which is the opposite side of the bidirectional relationship.

"Many" side is always the owning side of the relationship. In our case, the `User` has **many** `Tweets`, so `Tweet` is a "many" side, so it is an owning side of the relationship, and the `User` is the opposite one.

We can use a **many-to-one** relationship in the case of a unidirectional relationship, but usually, this type of relationship is applied for specifying a bidirectional connection. Let's see how to define a bidirectional relationship between `User` and `Tweet`.

The `Tweet` entity should have a reference to the `User` entity and vice versa. Let's define the `Tweet` entity field that refers to the `User` entity. We know that `Tweet` and `User` have the next relationship: many `Tweets` can belong to one `User` (many-to-one relationship), so you can use `@ManyToOne` annotation on top of the field for specifying such a relationship. Except for the type of relationship, we should define a mapped column (`UserID` column in the table `Tweet`) for joining an entity association using the `@JoinColumn` annotation.

```
@Entity
public class Tweet {

    @Id
    @Column(name = "TweetID")
    private long id;

    @ManyToOne
    @JoinColumn(name = "UserID")

    private User user;

}
```

Now we have to define another side of the relationship: the entity `User`. Once again: many `Tweets` can belong to one `User`; in other words, one `User` has many `Tweets`. It means that we can define the field `Tweets` with a type of `List<Tweet>` in the `User` entity. It helps us to refer to the `Tweet` entity using `@OneToMany` annotation with the `"mappedBy = "` parameter. This one indicates that the entity in this side (`User` entity) is the inverse of the relationship, and the owner resides in the "other" entity (`Tweet` entity).

```
@Entity
public class User {

    @Id
    private long id;

    @OneToMany(mappedBy = "user")
    private List<Tweet> tweets = new ArrayList<>();


}
```

The `mappedBy = "user"` parameter specifies that the `Tweet` entity `private User user;` field contains the foreign key to the user table so that we can find all tweets for the specified user.

§4. Cascade Operations

Now we have a bidirectional relationship between tweets and user entities, so each tweet knows about its user, and a user can refer to their tweets. Great! But let's imagine that we deleted a user, and from that moment, all their tweets should be deleted as well. Often entities that are in a relationship are dependent. It means that action that was performed on one entity should be executed for each entity that is dependent on it. As you can see, the tweets depend on the user, and the user removal entails the removal of all dependent tweets. Such operations are called **cascade operations**.

JPA presents six cascade types:

- `REMOVE`
- `PERSIST`
- `MERGE`
- `DETACH`
- `REFRESH`
- `ALL`

The cascade type is indicated in the annotation `@OneToMany` by `"cascade = "` parameter. All six enumerated cascade types are defined in the `javax.persistence.CascadeType` enum. Let's see an example with the `REMOVE` cascade type. This one indicates that if the parent entity (`User`) is removed from the current persistence context (specified row is deleted from the table `Users`), the related entities (all related `Tweets`) will also be removed.

```
@Entity
public class User {

    @Id
    private long id;

    @OneToMany(mappedBy = "user", cascade = CascadeType.REMOVE)
    private List<Tweet> tweets = new ArrayList<>();


}
```

The name of the CascadeType indicates which type of action would be shared between a parent and a child entity. `PERSIST` cascade type indicates that if the parent entity is saved (persisted) to the database table, the related entity will also be saved to the corresponding database table. `MERGE` cascade type indicates that if the parent entity is updated (merged), the related entity will also be merged. `DETACH` cascade type indicates that if the parent entity is detached (parent is still stored in the database table, but not managed by JPA anymore), the related entity will also be detached. `REFRESH` cascade type indicates that if the parent entity is refreshed (re-read from the database), the related entity will also be refreshed. `ALL` applies all cascade types.

What if we want to use not all, but only two cascade types, for example, `REMOVE` and `PERSIST`? Just indicate them in the list of comma-separated properties.

```
@Entity
public class User {

    @Id
    private long id;


 @OneToMany(mappedBy = "user", cascade = {CascadeType.REMOVE, CascadeType.PERSIST})
    private List<Tweet> tweets = new ArrayList<>();


}
```

§5. Conclusion

JPA can manage not only one entity but also a relationship between entities. There are four types of relationships between entities. However, **one-to-many** and **many-to-one** are the most popular types. You have learned how to deal with the unidirectional relationship by using `@ManyToOne` and `@JoinColumn` annotations. Now you also know how to deal with the bidirectional relationships by using `@OneToMany` annotation with the `"mappedBy = "` parameter. Entities can be dependent on each other, and we can handle dependencies by cascade operations. JPA has six types of cascade operations, and you have learned how to use them together with `@OneToMany` annotation and `"cascade = "` parameter that specifies the type of cascade operations.

Report a typo

65 users liked this theory. **8** didn't like it. **What about you?**

Start practicing

Comments (15) Hints (0) Useful links (1) Show discussion