The best way to map a @OneToMany relationship with JPA and Hibernate

Last modified: Nov 26, 2021

Imagine having a tool that can automatically detect JPA and Hibernate performance issues. Wouldn't that be just awesome?

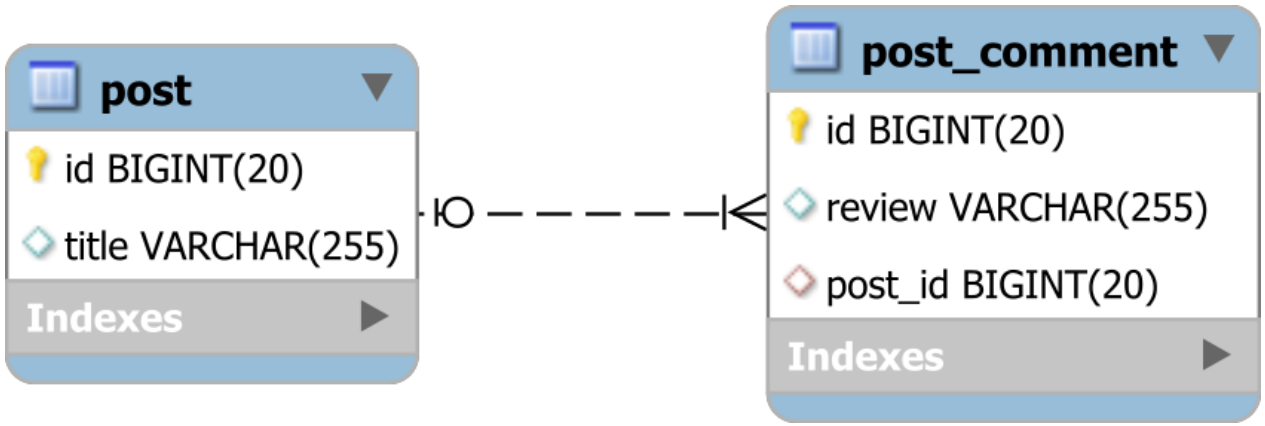
Well, Hypersistence Optimizer is that tool! And it works with Spring Boot, Spring Framework, Jakarta EE, Java EE, Quarkus, or Play Framework.

So, enjoy spending your time on the things you love rather than fixing performance issues in your production system on a Saturday night!

Introduction

statements is definitely not a trivial thing to do.

In a relational database system, a *one-to-many* association links two tables based on a Foreign Key column so that the child table record references the Primary Key of the parent table row.



As straightforward as it might be in a relational database, when it comes to JPA, the *one-to-many* database association can be represented either through a `@ManyToOne` or a `@OneToMany` association since the OOP association can be either unidirectional or bidirectional.

The `@ManyToOne` annotation allows you to map the Foreign Key column in the child entity mapping so that the child has an entity object reference to its parent entity. This is

For convenience, to take advantage of the entity state transitions and the dirty checking mechanism, many developers choose to map the child entities as a collection in the parent object, and, for this purpose, JPA offers the `@OneToMany` annotation.

As I explained in my book, many times, you are better off replacing collections with a query, which is much more flexible in terms of fetching performance. However, there are times when mapping a collection is the right thing to do, and then you have two choices:

- a unidirectional `@OneToMany` association
- a bidirectional `@OneToMany` association

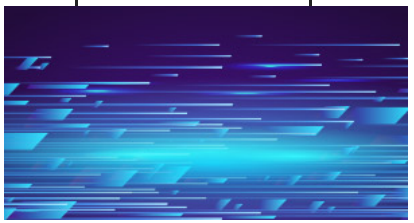
The bidirectional association requires the child entity mapping to provide a `@ManyToOne` annotation, which is responsible for controlling the association.

On the other hand, the unidirectional `@OneToMany` association is simpler since it's just the parent-side that defines the relationship. In this article, I'm going to explain the caveats of `@OneToMany` associations, and how you can overcome them.

There are many ways to map the `@OneToMany` association. We can use a List or a Set. We can also define the `@JoinColumn` annotation too. So, let's see how all this works.

Unidirectional @OneToMany

```java
@Entity(name = "Post")
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    @OneToMany(
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<PostComment> comments = new ArrayList<>();

    //Constructors, getters and setters removed for brevity
}

@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {

    @Id
    @GeneratedValue
    private Long id;

    private String review;

    //Constructors, getters and setters removed for brevity
}
```



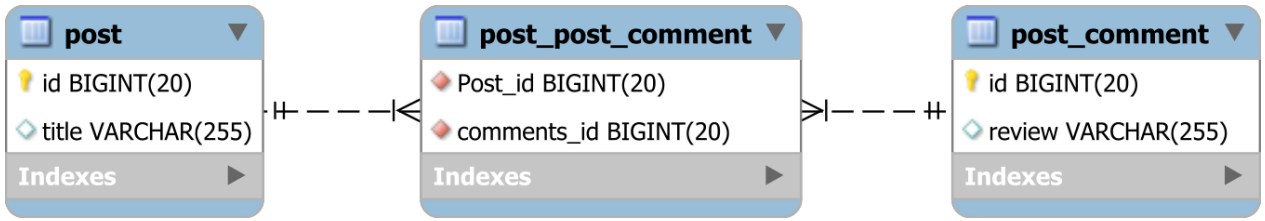
Now, if we persist one `Post` and three `PostComment(s)`:

```
    new PostComment("My first review")
);
post.getComments().add(
    new PostComment("My second review")
);
post.getComments().add(
    new PostComment("My third review")
);

entityManager.persist(post);
```


Hibernate is going to execute the following SQL statements:

```
insert into post (title, id)
values ('First post', 1)

insert into post_comment (review, id)
values ('My first review', 2)

insert into post_comment (review, id)
values ('My second review', 3)

insert into post_comment (review, id)
values ('My third review', 4)

insert into post_post_comment (Post_id, comments_id)
values (1, 2)

insert into post_post_comment (Post_id, comments_id)
values (1, 3)

insert into post_post_comment (Post_id, comments_id)
values (1, 4)
```

Well, by default, that's how the unidirectional `@OneToMany` association works, and this is how it looks from a database perspective:



For a DBA, this looks more like a *many-to-many* database association than a *one-to-many* relationship, and it's not very efficient either. Instead of two tables, we now have three tables, so we are using more storage than necessary. Instead of only one Foreign Key, we now have two of them. However, since we are most likely going to index these Foreign Keys, we are going to require twice as much memory to cache the index for this association. Not nice!

Unidirectional @OneToMany with @JoinColumn

To fix the aforementioned extra join table issue, we just need to add the `@JoinColumn` in the mix:

The `@JoinColumn` annotation helps Hibernate (the most famous JPA provider) to figure out that there is a `post_id` Foreign Key column in the `post_comment` table that defines this association.

With this annotation in place, when persisting the three `PostComment` entities, we get the following SQL output:

```
insert into post (title, id)
values ('First post', 1)

insert into post_comment (review, id)
values ('My first review', 2)

insert into post_comment (review, id)
values ('My second review', 3)

insert into post_comment (review, id)
values ('My third review', 4)

update post_comment set post_id = 1 where id = 2

update post_comment set post_id = 1 where id =  3

update post_comment set post_id = 1 where id =  4
```

A little bit better, but what's the purpose of those three update statements?

first without the Foreign Key since the child entity does not store this information. During the collection handling phase, the Foreign Key column is updated accordingly.

The same logic applies to collection state modifications, so when removing the firsts entry from the child collection:

```
post.getComments().remove(0);
```

Hibernate executes two statements instead of one:

```
update post_comment set post_id = null where post_id = 1 and id = 2

delete from post_comment where id=2
```

Again, the parent entity state change is executed first, which triggers the child entity update. Afterward, when the collection is processed, the orphan removal action will execute the child row delete statement.

So, is a `java.util.Set` any different?

Bidirectional @OneToMany

The best way to map a `@OneToMany` association is to rely on the `@ManyToOne` side to propagate all entity state changes:

```
@Entity(name = "Post")
@Table(name = "post")
public class Post {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    @OneToMany(
        mappedBy = "post",
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<PostComment> comments = new ArrayList<>();

    //Constructors, getters and setters removed for brevity

    public void addComment(PostComment comment) {
        comments.add(comment);
        comment.setPost(this);
    }

```

```
    }
}

@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {

    @Id
    @GeneratedValue
    private Long id;

    private String review;

    @ManyToOne(fetch = FetchType.LAZY)
    private Post post;

    //Constructors, getters and setters removed for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof PostComment )) return false;
        return id != null && id.equals(((PostComment) o).getId());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }
}
```

There are several things to note on the aforementioned mapping:

- The parent entity, `Post`, features two utility methods (e.g. `addComment` and `removeComment`) which are used to synchronize both sides of the bidirectional association. You should always provide these methods whenever you are working with a bidirectional association as, otherwise, you risk very subtle state propagation issues.
- The child entity, `PostComment`, implement the `equals` and `hashCode` methods. Since we cannot rely on a natural identifier for equality checks, we need to use the entity identifier instead for the `equals` method. However, you need to do it properly so that equality is consistent across all entity state transitions, which is also the reason why the `hashCode` has to be a constant value. Because we rely on equality for the `removeComment`, it's good practice to override `equals` and `hashCode` for the child entity in a bidirectional association.

If we persist three `PostComment(s)`:

```
Post post = new Post("First post");

post.addComment(
    new PostComment("My first review")
);
post.addComment(
    new PostComment("My second review")
);
post.addComment(
    new PostComment("My third review")
);

```

```
insert into post (title, id)
values ('First post', 1)

insert into post_comment (post_id, review, id)
values (1, 'My first review', 2)

insert into post_comment (post_id, review, id)
values (1, 'My second review', 3)

insert into post_comment (post_id, review, id)
values (1, 'My third review', 4)
```

If we remove a `PostComment`:

```
Post post = entityManager.find( Post.class, 1L );
PostComment comment1 = post.getComments().get( 0 );

post.removeComment(comment1);
```

There's only one delete SQL statement that gets executed:

```
delete from post_comment where id = 2
```

So, the bidirectional `@OneToMany` association is the best way to map a *one-to-many* database relationship when we really need the collection on the parent side of the association.

I also published a YouTube video about the Bidirectional @OneToMany association, so enjoy watching it if you're interested in this topic.

@ManyToOne might be just enough

Just because you have the option of using the `@OneToMany` annotation, it does not mean this should be the default option for every *one-to-many* database relationship. The

> Therefore, in reality, `@OneToMany` is practical only when many means few. Maybe `@OneToFew` would have been a more suggestive name for this annotation.

As I explained in this StackOverflow answer, you cannot limit the size of a `@OneToMany` collection like it would be the case if you used query-level pagination.

Therefore, most of the time, the `@ManyToOne` annotation on the child side is everything you need. But then, how do you get the child entities associated with a `Post` entity?

Well, all you need is just a single JPQL query:

```
List<PostComment> comments = entityManager.createQuery(
    "select pc " +
    "from PostComment pc " +
    "where pc.post.id = :postId", PostComment.class)
.setParameter( "postId", 1L )
.getResultList();
```

Which translates to a straightforward SQL query:

```
select pc.id AS id1_1_,
       pc.post_id AS post_id3_1_,
```

Even if the collection is not managed anymore, it's rather trivial to just add/remove child entities whenever necessary. As for updating child objects, the dirty checking mechanism works just fine even if you don't use a managed collection. What's nice about using a query is that you can paginate it any way you like so that, if the number of child entities grows with time, the application performance is not going to be affected.

Conclusion

SQL statements.

But then, even if they are very convenient, you don't always have to use collections. The `@ManyToOne` association is the most natural and also efficient way of mapping a *one-to-many* database relationship.

Category: Hibernate Tags: @JoinColumn, @ManyToOne, @OneToMany, association, bidirectional, hibernate, jpa, one-to-many, relationship, Training, Tutorial, unidirectional

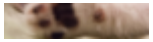
← How to find which statement failed in a JDBC Batch Update

Why you should use the Hibernate ResultTransformer to customize result set mappings →

10 Comments on “The best way to map a @OneToMany relationship with JPA and Hibernate”

Cat Thanh

March 5, 2023



” Again, the parent entity state change is executed first, which triggers the child entity update. Afterward, when the collection is processed, the orphan removal action will execute the child row delete statement.”
Thanks for the insightful article.

Reply

---

vladmihalcea
March 5, 2023

The orphan removal can trigger both a `DeleteAction` or an `OrphanRemovalAction`. The latter happens when the `DeleteEvent` has the `orphanRemovalBeforeUpdates` set to `true`.

Reply

Kostyantyn Panchenko
February 28, 2023

I’ve bookmarked this article and I used it a few times so far to refresh my knowledge. Also I’ve bought the book and I find it one of the most valuable books I’ve read.

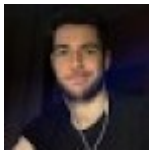
vladmihalcea

February 28, 2023

Thanks and stay tuned for more.

Reply

Dimitris



February 1, 2023

What would be the most efficient way to return the newly created entity?
Let’s say you create a new comment and you want to return it to the FE as you often do in a RESTful API. Since you’re saving using the parent entity (post) what would be the most efficient way to retrieve the comment you just created? For this use-case, is it better to directly save the comment with the entity manager?

Reply

vladmihalcea

February 1, 2023

In your example, the most efficient way to save the entity is to call `persist` on the child entity. You don't really need to use cascading unless the parent already fetched the child entities because it needs them further.

Reply

---

yahya bahhouss

December 24, 2022

Thank you for this nice article

Reply

---

vladmihalcea

December 25, 2022

You're welcome

Reply

Abhishek Omprakash

November 24, 2022

handle the scenario of deleting a post. Obviously one would want to that all the comments related to the post are also removed when deleting the post. (Note we are not using @OneToMany annotation in the Post entity class) What approach is well suited?
Thanks

Reply

vladmihalcea
November 24, 2022

A `CustomPostRepository`, which overrides the `delete` method and issues a bulk delete against the children. That's a very efficient option.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Before posting the comment, please take the time to read the FAQ page

Name *

Email *

Website

Notify me of follow-up comments by email.

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Hibernate
SQL
Spring
Git
FlexyPool

Twitter
Facebook
YouTube
GitHub
LinkedIn

About
FAQ
Archive
Privacy Policy
Terms of Service

Log in
Entries feed
Comments feed
WordPress.org