Hibernate Tip: Map a bidirectional many-to-many association

Question:

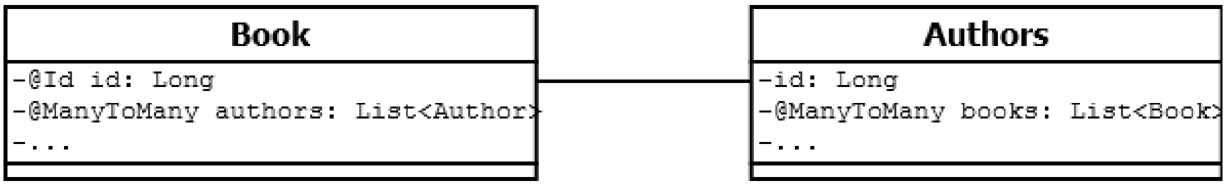
My table model contains a many-to-many association. How can I model it with Hibernate so that I can navigate it in both directions?

Solution:

You need to model the association on both entities if you want to be able to navigate it in both directions. Let’s have a look at an example.

Multiple Authors can write multiple books and a book can be written by one or more authors. That’s a typical many-to-many association, and you probably want to navigate it in both directions in your domain model and queries.

You need to model it as a many-to-many association on the Book entity and the Author entity.


Book
-@Id id: Long
-@ManyToMany authors: List<Author>
-...
Authors
-id: Long
-@ManyToMany books: List<Book>
-...

Let’s begin with the Book entity which is the owning side of the association in this example. That means that it defines the association and the Author entity just references it.

The relationship definition consists of two mandatory and one optional part. The entity attribute List authors and the @ManyToMany annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The @JoinTable annotation is optional.

It allows you to define the name of the join table and foreign key columns that store the many-to-many association.

I use it in the following code snippet to set the name of the join table to book_author and the names of the foreign key columns to fk_book and fk_author. If you don't define the name yourself, Hibernate generates default table and column names. The default table name is the combination of both entity names.

In this example, it would be Book_Author. The foreign key column name is generated by combining the name of the association mapping attribute and the name of the primary key attribute of the entity. These would be books_id and authors_id in this example.

```
@Entity
public class Book {


        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "id", updatable = false,
                        nullable = false)
        private Long id;


        @ManyToMany
        @JoinTable(
        name = "book_author",
        joinColumns = { @JoinColumn(name = "fk_book") },
        inverseJoinColumns = {
                @JoinColumn(name = "fk_author")})
        private List<Author> authors = new ArrayList<Author>();


        ...
}
```

You also need to map the many-to-many association on the Author entity to make it bidirectional.

As you can see in the following code snippet, this is done in a similar way as on the Book entity.

You need an attribute that models the association and a @ManyToMany annotation.

In this example, it’s the List books attribute which I annotated with a @ManyToMany annotation. The association is already defined on the Book entity. You can therefore just reference the attribute on the Book entity in the mappedBy attribute and Hibernate will use the same definition.

```
@Entity
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "id",
                        updatable = false,
                        nullable = false)
        private Long id;

        @ManyToMany(mappedBy="authors")
        private List<Book> books = new ArrayList<Book>();

        ...
}
```

That’s all you need to do to define a bidirectional many-to-many association. You can now navigate it in both directions in your JPQL or Criteria API queries or on your domain objects.

```
b = em.find(Book.class, 1L);
List<Author> authors = b.getAuthors();
```

Bidirectional associations are easy to use in queries, but they also require an additional step when you persist a new entity.

You need to update the association on both sides when you add or remove an entity.

You can see an example of it in the following code snippet in which I first create a new Author entity and add the Book entity to the List of books.

And after that, I also need to add the new Author entity to the List of authors on the Book entity.

```
Book b = em.find(Book.class, 1L);

Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

a.getBooks().add(b);
b.getAuthors().add(a);

em.persist(a);
```

Updating the associations on both entities is an error-prone task. It’s, therefore, a good practice to provide helper methods for it.

```
@Entity
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "id", updatable = false,
                nullable = false)
        private Long id;

        @ManyToMany(mappedBy="authors")
        private List<Book> books = new ArrayList<Book>();

        public void addBook(Book b) {
                this.books.add(b);
                b.getAuthors().add(this);
        }

        public void removeBook(Book b) {
                this.books.remove(b);
                b.getAuthors().remove(this);
        }

        ...
}
```