Special thanks to all Thoughts on Java Supporters for supporting this article!

Ultimate Guide – Association Mappings with JPA and Hibernate

By Thorben Janssen — 13 Comments

Association mappings are one of the key features of JPA and Hibernate. They model the relationship between two database tables as attributes in your domain model. That allows you to easily navigate the associations in your domain model and JPQL or Criteria queries.

JPA and Hibernate support the same associations as you know from your relational database model. You can use:

- one-to-one associations,
- many-to-one associations and
- many-to-many associations.

You can map each of them as a uni- or bidirectional association. That means you can either model them as an attribute on only one of the associated entities or on both. That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and JPQL or Criteria queries. I will explain that in more details in the first example.

Many-to-One Associations

An order consists of multiple items, but each item belongs to only one order. That is a typical example for a many-to-one association. If you want to model this in your database model, you need to store the primary key of the *Order* record as a foreign key in the *OrderItem* table.

With JPA and Hibernate, you can model this in 3 different ways. You can either model it as a bidirectional association with an attribute on the *Order* and the *OrderItem* entity. Or you can model it as a unidirectional relationship with an attribute on the *Order* or the *OrderItem* entity.

Unidirectional Many-to-One Association

Let’s take a look at the unidirectional mapping on the *OrderItem* entity first. The *OrderItem* entity represents the many side of the relationship and the *OrderItem* table contains the foreign key of the record in the *Order* table.

As you can see in the following code snippet, you can model this association with an attribute of type *Order* and a *@ManyToOne* annotation. The *Order order* attribute models the association, and the annotation tells Hibernate how to map it to the database.

```
@Entity
public class OrderItem {

    @ManyToOne
    private Order order;

    …
}
```

That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column with the name *order_id* to store the foreign key to the *Order* entity.

If you want to use a different column, you need to define the foreign key column name with a *@JoinColumn* annotation. The example in the following code snippet tells Hibernate to use the column *fk_order* to store the foreign key.

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = “fk_order”)
    private Order order;

    …
}
```

You can now use this association in your business code to get the *Order* for a given *OrderItem* and to add or remove an *OrderItem* to or from an existing *Order*.

```
Order o = em.find(Order.class, 1L);

OrderItem i = new OrderItem();
i.setOrder(o);

em.persist(i);
```

That’s all about the mapping of unidirectional many-to-one associations for now. If you want to dive deeper, you should take a look at *FetchTypes*. I explained them in detail in my Introduction to JPA FetchTypes.

But now, let’s continue with the association mappings and talk about unidirectional one-to-many relationships next. As you might expect, the mapping is very similar to this one.

Already a member? Login here.

Unidirectional One-to-Many Association

The unidirectional one-to-many relationship mapping is not very common. In the example of this post, it only models the association on the *Order* entity and not on the *OrderItem*.

The basic mapping definition is very similar to the many-to-one association. It consist of the *List items* attribute which stores the associated entities and a *@OneToMany* association.

```
@Entity
public class Order {

    @OneToMany
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

But this is most likely not the mapping you're looking for because Hibernate uses an association table to map the relationship. If you want to avoid that, you need to use a *@JoinColumn* annotation to specify the foreign key column.

The following code snippet shows an example of such a mapping. The *@JoinColumn* annotation tells Hibernate to use the *fk_order* column in the *OrderItem* table to join the two database tables.

```
@Entity
public class Order {

    @OneToMany
    @JoinColumn(name = "fk_order")
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

You can now use the association in your business code to get all *OrderItem*s of an *Order* and to add or remove an *OrderItem* to or from an *Order*.

```
Order o = em.find(Order.class, 1L);

OrderItem i = new OrderItem();

o.getItems().add(i);

em.persist(i);
```

Bidirectional Many-to-One Associations

The bidirectional Many-to-One association mapping is the most common way to model this relationship with JPA and Hibernate. It uses an attribute on the *Order* and the *OrderItem* entity. This allows you to navigate the association in both directions in your domain model and your JPQL queries.

The mapping definition consists of 2 parts:

- the to-many side of the association which owns the relationship mapping and
- the to-one side which just references the mapping

Let's take a look at the owning side first. You already know this mapping from the unidirectional Many-to-One association mapping. It consists of the *Order order* attribute, a *@ManyToOne* annotation and an optional *@JoinColumn* annotation.

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = “fk_order”)
    private Order order;

    …
}
```

The owning part of the association mapping already provides all the information Hibernate needs to map it to the database. That makes the definition of the referencing part simple. You just need to reference the owning association mapping. You can do that by providing the name of the association-mapping attribute to the *mappedBy* attribute of the *@OneToMany* annotation. In this example, that’s the *order* attribute of the *OrderItem* entity.

```
@Entity
public class Order {

    @OneToMany(mappedBy = “order”)
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

You can now use this association in a similar way as the unidirectional relationships I showed you before. But adding and removing an entity from the relationship requires an additional step. You need to update both sides of the association.

```
Order o = em.find(Order.class, 1L);

OrderItem i = new OrderItem();
i.setOrder(o);

o.getItems().add(i);

em.persist(i);
```

That is an error-prone task, and a lot of developers prefer to implement it in a utility method which updates both entities.

```
@Entity
public class Order {
    …

    public void addItem(OrderItem item) {
        this.items.add(item);
        item.setOrder(this);
    }
    …
}
```

That’s all about many-to-one association mapping for now. You should also take a look at *FetchTypes* and how they impact the way Hibernate loads entities from the database. I get into detail about them in my Introduction to JPA FetchTypes.

Many-to-Many Associations

Many-to-Many relationships are another often used association type. On the database level, it requires an additional association table which contains the primary key pairs of the associated entities. But as you will see, you don't need to map this table to an entity.

A typical example for such a many-to-many association are *Product*s and *Store*s. Each *Store* sells multiple *Product*s and each *Product* gets sold in multiple *Store*s.

Similar to the many-to-one association, you can model a many-to-many relationship as a uni- or bidirectional relationship between two entities.

But there is an important difference that might not be obvious when you look at the following code snippets. When you map a many-to-many association, you should use a *Set* instead of a *List* as the attribute type. Otherwise, Hibernate will take a very inefficient approach to remove entities from the association. It will remove all records from the association table and re-insert the remaining ones. You can avoid that by using a *Set* instead of a *List* as the attribute type.

OK let's take a look at the unidirectional mapping first.

Unidirectional Many-to-Many Associations

Similar to the previously discussed mappings, the unidirectional many-to-many relationship mapping requires an entity attribute and a *@ManyToMany* annotation. The attribute models the association and you can use it to navigate it in your domain model or JPQL queries. The annotation tells Hibernate to map a many-to-many association.

Let's take a look at the relationship mapping between a *Store* and a *Product*. The *Set products* attribute models the association in the domain model and the *@ManyToMany* association tells Hibernate to map it as a many-to-many association.

And as I already explained, please note the difference to the previous many-to-one mappings. You should map the associated entities to a *Set* instead of a *List.*

```
@Entity
public class Store {

   @ManyToMany
   private Set<Product> products = new HashSet<Product>();

   …
}
```

If you don't provide any additional information, Hibernate uses its default mapping which expects an association table with the name of both entities and the primary key attributes of both entities. In this case, Hibernate uses the *Store_Product* table with the columns *store_id* and *product_id*.

You can customize that with a *@JoinTable* annotation and its attributes *joinColumns* and *inverseJoinColumns*. The *joinColumns* attribute defines the foreign key columns for the entity on which you define the association mapping. The *inverseJoinColumns* attribute specifies the foreign key columns of the associated entity.

The following code snippet shows a mapping that tells Hibernate to use the *store_product* table with the *fk_product* column as the foreign key to the *Product* table and the *fk_store* column as the foreign key to the *Store* table.

```
@Entity
public class Store {

  @ManyToMany
  @JoinTable(name = “store_product”,
         joinColumns = { @JoinColumn(name = “fk_store”) },
         inverseJoinColumns = { @JoinColumn(name = “fk_product”) })
  private Set<Product> products = new HashSet<Product>();

  …
}
```

That’s all you have to do to define an unidirectional many-to-many association between two entities. You can now use it to get a *Set* of associated entities in your domain model or to join the mapped tables in a JPQL query.

```
Store s = em.find(Store.class, 1L);

Product p = new Product();

s.getProducts().add(p);

em.persist(p);
```

Bidirectional Many-to-Many Associations

The bidirectional relationship mapping allows you to navigate the association in both directions. And after you’ve read the post this far, you’re probably not surprised when I tell you that the mapping follows the same concept as the bidirectional mapping of a many-to-one relationship.

One of the two entities owns the association and provides all mapping information. The other entity just references the association mapping so that Hibernate knows where it can get the required information.

Let’s start with the entity that owns the relationship. The mapping is identical to the unidirectional many-to-many association mapping. You need an attribute that maps the association in your domain model and a *@ManyToMany* association. If you want to adapt the default mapping, you can do that with a *@JoinColumn* annotation.

```
@Entity
public class Store {

  @ManyToMany
  @JoinTable(name = “store_product”,
         joinColumns = { @JoinColumn(name = “fk_store”) },
         inverseJoinColumns = { @JoinColumn(name = “fk_product”) })
  private Set<Product> products = new HashSet<Product>();

  …
}
```

The mapping for the referencing side of the relationship is a lot easier. Similar to the bidirectional many-to-one relationship mapping, you just need to reference the attribute that owns the association.

You can see an example of such a mapping in the following code snippet. The *List products* attribute of the *Store* entity owns the association. So, you only need to provide the *String* "*products*" to the *mappedBy* attribute of the *@ManyToMany* annotation.

```
@Entity
public class Product{

    @ManyToMany(mappedBy="products")
    private Set<Store> stores = new HashSet<Store>();

    …
}
```

That's all you need to do to define a bidirectional many-to-many association between two entities. But there is another thing you should do to make it easier to use the bidirectional relationship.

You need to update both ends of a bidirectional association when you want to add or remove an entity. Doing that in your business code is verbose and error-prone. It's, therefore, a good practice to provide helper methods which update the associated entities.

```
@Entity
public class Store {

    public void addProduct(Product p) {
        this.products.add(p);
        p.getStores().add(this);
    }

    public void removeProduct(Product p) {
        this.products.remove(p);
        p.getStores().remove(this);
    }

    …
}
```

OK, now we're done with the definition of the many-to-many association mappings. Let's take a look at the third and final kind of association: The one-to-one relationship.

One-to-One Associations

One-to-one relationships are rarely used in relational table models. You, therefore, won't need this mapping too often. But you will run into it from time to time. So it's good to know that you can map it in a similar way as all the other associations.

An example for a one-to-one association could be a *Customer* and the *ShippingAddress*. Each *Customer* has exactly one *ShippingAddress* and each *ShippingAddress* belongs to one *Customer*. On the database level, this mapped by a foreign key column either on the *ShippingAddress* or the *Customer* table.

Let's take a look at the unidirectional mapping first.

Unidirectional One-to-One Associations

As in the previous unidirectional mapping, you only need to model it for the entity for which you want to navigate the relationship in your query or domain model. Let’s say you want to get from the *Customer* to the *ShippingAddress* entity.

The required mapping is similar to the previously discussed mappings. You need an entity attribute that represents the association, and you have to annotate it with an *@OneToOne* annotation.

When you do that, Hibernate uses the name of the associated entity and the name of its primary key attribute to generate the name of the foreign key column. In this example, it would use the column *shippingaddress_id*. You can customize the name of the foreign key column with a *@JoinColumn* annotation. The following code snippet shows an example of such a mapping.

```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```

That’s all you need to do to define a one-to-one association mapping. You can now use it in your business to add or remove an association, to navigate it in your domain model or to join it in a JPQL query.

```
Customer c = em.find(Customer.class, 1L);
ShippingAddress sa = c.getShippingAddress();
```

Bidirectional One-to-One Associations

The bidirectional one-to-one relationship mapping extends the unidirectional mapping so that you can also navigate it in the other direction. In this example, you also model it on the *ShippingAddress* entity so that you can get the *Customer* for a giving *ShippingAddress*.

Similar to the previously discussed bidirectional mappings, the bidirectional one-to-one relationship consists of an owning and a referencing side. The owning side of the association defines the mapping, and the referencing one just links to that mapping.

The definition of the owning side of the mapping is identical to the unidirectional mapping. It consists of an attribute that models the relationship and is annotated with a *@OneToOne* annotation and an optional *@JoinColumn* annotation.

```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```

The referencing side of the association just links to the attribute that owns the relationship. Hibernate gets all information from the referenced mapping, and you don’t need to provide any additional information. You can define that with the *mappedBy* attribute of the *@OneToOne* annotation. The following code snippet shows an example of such a mapping.

```
@Entity
public class ShippingAddress{

    @OneToOne(mappedBy = “shippingAddress”)
    private Customer customer;

    …
}
```

Summary

The relational table model uses many-to-many, many-to-one and one-to-one associations to model the relationship between database records. You can map the same relationships with JPA and Hibernate, and you can do that in an unidirectional or bidirectional way.

The unidirectional mapping defines the relationship only on 1 of the 2 associated entities, and you can only navigate it in that direction. The bidirectional mapping models the relationship for both entities so that you can navigate it in both directions.

The concept for the mapping of all 3 kinds of relationships is the same.

If you want to create an unidirectional mapping, you need an entity attribute that models the association and that is annotated with a *@ManyToMany*, *@ManyToOne*, *@OneToMany* or *@OneToOne* annotation. Hibernate will generate the name of the required foreign key columns and tables based on the name of the entities and their primary key attributes.

The bidirectional associations consist of an owning and a referencing side. The owning side of the association is identical to the unidirectional mapping and defines the mapping. The referencing side only links to the attribute that owns the association.

Related Posts:

1. Getting Started With Hibernate
2. Hibernate Tips: How to delete child entities from a many-to-one association
3. Hibernate Tips: How to model an association that doesn’t reference primary key columns
4. A Beginner’s Guide to JPA’s persistence.xml

Become a Thoughts on Java Supporter to claim your member perks and to help me write more articles like this.

Filed Under: Featured, Hibernate Beginner, JPA
Tagged With: Association Mapping, Mapping

Implement Your Persistence Layer with Ease

Learn More About Hibernate

Need Some Help with Your Project?

Comments

Thành Loyal says
May 7, 2017 at 8:22 am

Hello Thorben,
Could you please provide some guide on how to remove one or some items from items list of a order?
Thank you!

Reply

lee Shen says
May 14, 2017 at 1:05 pm

excellent resourse

Reply

Sovello says
November 30, 2017 at 4:11 pm

Thanks a million times. So easily explained and so easy to understand

Reply

Thorben Janssen says
December 1, 2017 at 5:36 pm

Thanks, for reading it, Sovello 🙂

Reply

David says
February 15, 2018 at 5:32 pm

Is it a typo in generic and there should be Product?
@Entity
public class Store {

@ManyToMany
private Set products = new HashSet();

…
}

Reply

Thorben Janssen says
April 22, 2018 at 9:47 am

Yes, that's a typo. Fixed it. Thanks!

Reply

Lukasz Mielczarek says
April 5, 2018 at 11:04 pm

Great guide! Much easier than official Oracle's JPA tutorial 🙂
I've noticed just one flaw in Unidirectional Many-to-Many Associations section :
@Entity
public class Store {

@ManyToMany
private Set products = new HashSet();

…
}

Shouldn't be there Set products = new HashSet(); 😛 ?

Reply

Thorben Janssen says
April 21, 2018 at 2:55 pm

Thanks, Lukasz.
You're right, of course. I fixed it 🙂

Reply

Sushant says
April 30, 2018 at 12:40 am

A really good, easy to understand tutorial!

Reply

Nikhil Kapoor says
July 20, 2018 at 4:50 pm

HI
Great article.
If I have a requirement similar to country state city with top-down one-to-many which mapping methods should I use?

Reply

Thorben Janssen says
July 21, 2018 at 6:54 pm

That depends on the number of elements on the many-site of your associations.
If your database contains several thousand cities for each state, then you should use a unidirectional many-to-one association. I would not model it as a bidirectional association because fetching several thousand entities will create performance issues.
If the many-site doesn’t contain that many elements, I would use a bidirectional many-to-one association.

Reply

john says
March 8, 2019 at 9:10 pm

In above topic : Bidirectional One-to-One Associations
Customer is owning side and refrence is also created in cutomer table , is it a preferred way or we should store customer_id in shippingAddress table.

Reply

Thorben Janssen says
March 12, 2019 at 2:23 pm

You need to model the owning side of the association on the entity that maps the database table that contains the foreign key column. For a one-to-one association, it doesn’t matter on which side you model the foreign key.

Reply

Leave a Reply

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

Comment

Name *

Email *

Website

☐

Save my name, email, and website in this browser for the next time I comment.

POST
COMMEN

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

LET'S CONNECT

Thorben Janssen

Independent consultant, trainer and author

SPEAKING AT

7th February 2019

JUG Ostfalen (Germany):

Hibernate Tips 'n' Tricks: Typische Probleme schnell gelöst (Talk - German)

19th March 2019

JavaLand 2019 (Germany):

Hibernate Tips 'n' Tricks: Typische Probleme schnell gelöst (Talk - German)

21st March 2019

JavaLand 2019 (Germany):

Hibernate + jOOQ + Flyway = Die besten Frameworks in einem Stack (1-day Workshop - German)

6th-10th May 2019

JAX 2019 (Germany):

Hibernate Workshop: Komplexe Lösungen jenseits von CRUD (2-day Workshop - German)

6th-10th May 2019

JAX 2019 (Germany):

Spring Data JDBC vs. Spring Data JPA – Wer macht das Rennen? (Talk - German)

6th-10th May 2019

JAX 2019 (Germany):

Microservices mit Hibernate – typische Probleme und Lösungen (Talk - German)

Looking for an on-site training?

F E A T U R E D   P O S T

14 YouTube Channels You Should Follow in 2019

Getting Started With Hibernate

Entities or DTOs – When should you use which projection?

Ultimate Guide – Association Mappings with JPA and Hibernate

Ultimate Guide to JPQL Queries with JPA and Hibernate

R E C E N T   P O S T S

Hibernate Tips: How to override column mappings of a superclass

Updated JPA for Beginners Training and Price Increase Warning

What is Spring Data JPA? And why should you use it?

Hibernate Tips: How to Customize a Constructor Expression for Different Subclasses

Hibernate Tips: How to Map java.time.Year with JPA and Hibernate

Why, When and How to Use DTO Projections with JPA and Hibernate

Hibernate Tip: Map a bidirectional one-to-one association with shared composite primary key

Implementing the Repository pattern with JPA and Hibernate

Hibernate Tips: How to Handle NULL Values while Ordering Query Results in a CriteriaQuery

6 Hibernate Best Practices for Readable and Maintainable Code