Ultimate Guide – Association Mappings with JPA and Hibernate

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**(https://thoughts-on-java.org/entity-mappings-introduction-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.

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 OrderItems 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 Products and Stores. Each Store sells multiple Products and each Product gets sold in multiple Stores.

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 **inverse-JoinColumns**. 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.

Entity Mappings: Introduction to JPA FetchTypes

The FetchType defines when Hibernate gets the related entities from the database, and it is one of the crucial elements for a fast persistence tier. In general, you want to fetch the entities you use in your business tier as efficiently as possible. But that's not that easy. You either get all relationships with one query or you fetch only the root entity and initialize the relationships as soon as you need them.

I'll explain both approaches in more detail during this post and also provide you some links to more advanced solutions that combine flexibility and efficiency.

Default FetchTypes and how to change it

When you started with Hibernate, you most, likely either didn't know about FetchTypes or you were told to always use FetchType.LAZY. In general, that's a good recommendation. But what does it exactly mean? And what is the default if you don't define the FetchType?

The default depends on the cardinality of the relationship. All to-one relationships use FetchType.EAGER and all to-many relationships FetchType.LAZY.

Even the best default doesn't fit for all use cases, and you sometimes want to change it. You can do this by providing your preferred FetchType to the relationship annotation as you can see in the following code snippet.

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

  @OneToMany(mappedBy = "order", fetch = FetchType.EAGER)
  private Set<OrderItem> items = new HashSet<OrderItem>();
    ...
 }
```

FetchType.EAGER – Fetch it so you'll have it when you need it

The FetchType.EAGER tells Hibernate to get all elements of a relationship when selecting the root entity. As I explained earlier, this is the default for to-one relationships, and you can see it in the following code snippets.

I use the default FetchType (EAGER) for the many-to-one relationship between the OrderItem and Product entity.

```
@Entity
public class OrderItem implements Serializable {
  @ManyToOne
  private Product product;
  ...
}
```

When I now fetch an OrderItem entity from the database, Hibernate will also get the related Product entity.

```
OrderItem orderItem = em.find(OrderItem.class, 1L);
```

```
log.info("Fetched OrderItem: "+orderItem);
Assert.assertNotNull(orderItem.getProduct());
```

05:01:24,504 DEBUG SQL:92 - select orderitem0_.id as id1_0_0_, orderitem0_.order_id as order_id4_0_0_, orderitem0_.product_id as product_5_0_0_, orderitem0_.quantity as quantity2_0_0_, orderitem0_.version as version3_0_0_, order1_.id as id1_2_1_, order1_.orderNumber as orderNum2_2_1_, order1_.version as version3_2_1_, product2_.id as id1_1_2_, product2_.name as name2_1_2_, product2_.price as price3_1_2_, product2_.version as version4_1_2_ from OrderItem orderitem0_ left outer join purchaseOrder order1_ on orderitem0_.order_id=order1_.id left outer join Product product2_ on orderitem0_.product_id=product2_.id where orderitem0_.id=?
05:01:24,557 INFO FetchTypes:77 - Fetched OrderItem: OrderItem , quantity: 100

This seems to be very useful in the beginning. Joining the required entities and getting all of them in one query is very efficient.

But keep in mind, that Hibernate will ALWAYS fetch the Product entity for your OrderItem, even if you don't use it in your business code. If the related entity isn't too big, this is not an issue for to-one relationships. But it will most likely slow down your application if you use it for a to-many relationship that you don't need for your use case. Hibernate then has to fetch tens or even hundreds of additional entities which creates a significant overhead.

FetchType.LAZY – Fetch it when you need it

The FetchType.LAZY tells Hibernate to only fetch the related entities from the database when you use the relationship. This is a good idea in general because there's no reason to select entities you don't need for your uses case. You can see an example of a lazily fetched relationship in the following code snippets.

The one-to-many relationship between the Order and the OrderItem entities uses the default FetchType for to-many relationships which is lazy.

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

  @OneToMany(mappedBy = "order")
  private Set<OrderItem> items = new HashSet<OrderItem>();
   ...
}
```

The used FetchType has no influence on the business code. You can call the getOrderItems() method just as any other getter method.

```
Order newOrder = em.find(Order.class, 1L);
log.info("Fetched Order: "+newOrder);
Assert.assertEquals(2, newOrder.getItems().size());
```

Hibernate handles the lazy initialization transparently and fetches the OrderItem entities as soon as the getter method gets called.

05:03:01,504 DEBUG SQL:92 - select order0_.id as id1_2_0_, order0_.orderNumber as orderNum2_2_0_, order0_.version as version3_2_0_ from purchaseOrder order0_ where order0_.id=?
05:03:01,545 INFO FetchTypes:45 - Fetched Order: Order orderNumber: order1
05:03:01,549 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?

Handling lazy relationships in this way is perfectly fine if you work on a single Order entity or a small list of entities. But it becomes a performance problem when you do it on a large list of entities. As you can see in the following log messages, Hibernate has to perform an additional SQL statement for each Order entity to fetch its OrderItems.

05:03:40,936 DEBUG ConcurrentStatisticsImpl:411 - HHH000117: HQL: SELECT o FROM Order o, time: 41ms, rows: 3
05:03:40,939 INFO FetchTypes:60 - Fetched all Orders
05:03:40,942 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?
05:03:40,957 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?
05:03:40,959 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?

This behavior is called n+1 select issue, and it's the most common performance problem. It is so common that you most likely have it if you didn't explicitly search for it. If you're not sure how to do that, signup for my free, 3-part video course about finding and fixing n+1 select issues.

There are two ways to avoid these issues:

You can use FetchType.EAGER if you know that all of your use cases that fetch an Order entity also need to process the related OrderItem entities. That will almost never be the case.

If there are some use cases which only work on Order entities (which is most likely the case), you

should use FetchType.LAZY in your entity mapping and use one of these options to initialize the relationship when you need them.

Summary and cheat sheet

As I said in the beginning, you need to make sure to use the right FetchType for your use case to avoid common Hibernate performance issues. For most use cases, the FetchType.LAZY is a good choice. But make sure that you don't create any n+1 select issues.

Let's quickly summarize the different FetchTypes.

EAGER fetching tells Hibernate to get the related entities with the initial query. This can be very efficient because all entities are fetched with only one query. But in most cases it just creates a huge overhead because you select entities you don't need in your use case.

You can prevent this with FetchType.LAZY. This tells Hibernate to delay the initialization of the relationship until you access it in your business code. The drawback of this approach is that Hibernate needs to execute an additional query to initialize each relationship.