Association Mappings

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

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.

Unidirectional Many-to-One Association

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. You can use the optional *@JoinColumn* annotation to define the name of the foreign key column.

```
@Entity
public class OrderItem {

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

	…
}
```

Unidirectional One-to-Many Association

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.

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.

```
@Entity
public class Order {

	@OneToMany
	@JoinColumn(name = “fk_order”)
	private List items = new ArrayList();

	…
}
```

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

The mapping specification of the owning side is identical to the unidirectional mapping.

```
@Entity
public class OrderItem {

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

    …
}
```

The definition of the referencing part is a lot simpler. 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 items = new ArrayList();

        …
}
```

Adding and removing entities from a bidirectional association requires you to update both sides of the relationship. 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);
        }
        …
}
```

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.

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.

```
@Entity
public class Store {

    @ManyToMany
    @JoinTable(name = "store_product",
        joinColumns = {@JoinColumn(name = "fk_store")},
        inverseJoinColumns =
            {@JoinColumn(name = "fk_product")})
    private List<Product> products =
        new ArrayList<Product>();

    ...
}
```

You can customize the join table with the *@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.

Bidirectional Many-to-Many Associations

The bidirectional relationship mapping allows you to navigate the association in both directions. The association mapping consists of an owning and a referencing part. The owning part provides all mapping information and the referencing part only links to it.

The mapping of the owning side is identical to the unidirectional many-to-many association mapping.

```
@Entity
public class Store {

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

    …
}
```

The mapping for the referencing side of the relationship is a lot easier. You just need to reference the attribute that owns the association.

```
@Entity
public class Product{

        @ManyToMany(mappedBy="products")
        private List<Store> stores = new ArrayList<Store>();

        …
}
```

You need to update both ends of a bidirectional association when you want to add or remove an entity. It's 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);
        }
        …
}
```

One-to-One 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.

Unidirectional One-to-One Associations

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.

```
@Entity
public class Customer{

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

    …
}
```

Bidirectional One-to-One Associations

The bidirectional one-to-one relationship mapping extends the unidirectional mapping with an association-referencing side so that you can navigate it in both direction.

The definition of the owning side of the mapping is identical to the unidirectional mapping.

```
@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.

```
@Entity
public class ShippingAddress{

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

    …
}
```