# 5. Association Mapping

This chapter explains mapping associations between objects.

Instead of working with foreign keys in your code, you will always work with references to objects instead and Doctrine will convert those references to foreign keys internally.

- A reference to a single object is represented by a foreign key.
- A collection of objects is represented by many foreign keys pointing to the object holding the collection

This chapter is split into three different sections.

- A list of all the possible association mapping use-cases is given.
- Mapping Defaults are explained that simplify the use-case examples.
- Collections are introduced that contain entities in associations.

To gain a full understanding of associations you should also read about **owning and inverse sides of associations** (unitofwork-associations.html)

## 5.1. Many-To-One, Unidirectional

A many-to-one association is the most common association between objects.

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

/** @Entity */
class Address
{
    // ...
}
```

The above `@JoinColumn` is optional as it would default to `address_id` and `id` anyways. You can omit it and let it use the defaults.

Generated MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    address_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE Address (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE User ADD FOREIGN KEY (address_id) REFERENCES Address(id);
```

## 5.2. One-To-One, Unidirectional

Here is an example of a one-to-one association with a `Product` entity that references one `Shipping` entity. The `Shipping` does not reference back to the `Product` so that the reference is said to be unidirectional, in one direction only.

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class Product
{
    // ...

    /**
     * @OneToOne(targetEntity="Shipping")
     * @JoinColumn(name="shipping_id", referencedColumnName="id")
     */
    private $shipping;

    // ...
}

/** @Entity */
class Shipping
{
    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    shipping_id INT DEFAULT NULL,
    UNIQUE INDEX UNIQ_6FBC94267FE4B2B (shipping_id),
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Shipping (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Product ADD FOREIGN KEY (shipping_id) REFERENCES Shipping(id);
```

## 5.3. One-To-One, Bidirectional

Here is a one-to-one relationship between a `Customer` and a `Cart`. The `Cart` has a reference back to the `Customer` so it is bidirectional.

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class Customer
{
    // ...

    /**
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;

    // ...
}

/** @Entity */
class Cart
{
    // ...

    /**
     * @OneToOne(targetEntity="Customer", inversedBy="cart")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;

    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Cart (
    id INT AUTO_INCREMENT NOT NULL,
    customer_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Customer (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Cart ADD FOREIGN KEY (customer_id) REFERENCES Customer(id);
```

See how the foreign key is defined on the owning side of the relation, the table `Cart`.

## 5.4. One-To-One, Self-referencing

You can define a self-referencing one-to-one relationships like below.

```
<?php
/** @Entity */
class Student
{
    // ...

    /**
     * @OneToOne(targetEntity="Student")
     * @JoinColumn(name="mentor_id", referencedColumnName="id")
     */
    private $mentor;

    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

With the generated MySQL Schema:

```
CREATE TABLE Student (
    id INT AUTO_INCREMENT NOT NULL,
    mentor_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Student ADD FOREIGN KEY (mentor_id) REFERENCES Student(id);
```

## 5.5. One-To-Many, Bidirectional

A one-to-many association has to be bidirectional, unless you are using an additional join-table. This is necessary, because of the foreign key in a one-to-many association being defined on the “many” side. Doctrine needs a many-to-one association that defines the mapping of this foreign key.

This bidirectional mapping requires the `mappedBy` attribute on the `OneToMany` association and the `inversedBy` attribute on the `ManyToOne` association.

**PHP** **XML** **YAML**

```
<?php
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class Product
{
    // ...
    /**
     * @OneToMany(targetEntity="Feature", mappedBy="product")
     */
    private $features;
    // ...

    public function __construct() {
        $this->features = new ArrayCollection();
    }
}

/** @Entity */
class Feature
{
    // ...
    /**
     * @ManyToOne(targetEntity="Product", inversedBy="features")
     * @JoinColumn(name="product_id", referencedColumnName="id")
     */
    private $product;
    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Feature (
    id INT AUTO_INCREMENT NOT NULL,
    product_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Feature ADD FOREIGN KEY (product_id) REFERENCES Product(id);
```

## 5.6. One-To-Many, Unidirectional with Join Table



A unidirectional one-to-many association can be mapped through a join table. From Doctrine’s point of view, it is simply mapped as a unidirectional many-to-many whereby a unique constraint on one of the join columns enforces the one-to-many cardinality.

The following example sets up such a unidirectional one-to-many association:

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Phonenumber")
     * @JoinTable(name="users_phonenumbers",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id",
 unique=true)}
     *      )
     */
    private $phonenumbers;

    public function __construct()
    {
        $this->phonenumbers = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}

/** @Entity */
class Phonenumber
{
    // ...
}
```

Generates the following MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE users_phonenumbers (
    user_id INT NOT NULL,
    phonenumber_id INT NOT NULL,
    UNIQUE INDEX users_phonenumbers_phonenumber_id_uniq (phonenumber_id),
    PRIMARY KEY(user_id, phonenumber_id)
) ENGINE = InnoDB;

CREATE TABLE Phonenumber (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE users_phonenumbers ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE users_phonenumbers ADD FOREIGN KEY (phonenumber_id) REFERENCES Phonenumber(id);
```

## 5.7. One-To-Many, Self-referencing

You can also setup a one-to-many association that is self-referencing. In this example we setup a hierarchy of `Category` objects by creating a self referencing relationship. This effectively models a hierarchy of categories and from the database perspective is known as an adjacency list approach.

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class Category
{
    // ...
    /**
     * @OneToMany(targetEntity="Category", mappedBy="parent")
     */
    private $children;

    /**
     * @ManyToOne(targetEntity="Category", inversedBy="children")
     * @JoinColumn(name="parent_id", referencedColumnName="id")
     */
    private $parent;
    // ...
    public function __construct() {
        $this->children = new \Doctrine\Common\Collections\ArrayCollection();
    }
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Category (
    id INT AUTO_INCREMENT NOT NULL,
    parent_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Category ADD FOREIGN KEY (parent_id) REFERENCES Category(id);
```

## 5.8. Many-To-Many, Unidirectional

Real many-to-many associations are less common. The following example shows a unidirectional association between User and Group entities:

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id", referencedColumnName="id")}
     *      )
     */
    private $groups;

    // ...

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

/** @Entity */
class Group
{
    // ...
}
```

Generated MySQL Schema:

```
    CREATE TABLE User (
        id INT AUTO_INCREMENT NOT NULL,
        PRIMARY KEY(id)
    ) ENGINE = InnoDB;
    CREATE TABLE users_groups (
        user_id INT NOT NULL,
        group_id INT NOT NULL,
        PRIMARY KEY(user_id, group_id)
    ) ENGINE = InnoDB;
    CREATE TABLE Group (
        id INT AUTO_INCREMENT NOT NULL,
        PRIMARY KEY(id)
    ) ENGINE = InnoDB;
    ALTER TABLE users_groups ADD FOREIGN KEY (user_id) REFERENCES User(id);
    ALTER TABLE users_groups ADD FOREIGN KEY (group_id) REFERENCES Group(id);
```

> Why are many-to-many associations less common? Because frequently you want to associate additional attributes with an association, in which case you introduce an association class. Consequently, the direct many-to-many association disappears and is replaced by one-to-many/many-to-one associations between the 3 participating classes.

## 5.9. Many-To-Many, Bidirectional

Here is a similar many-to-many relationship as above except this one is bidirectional.

**PHP** **XML** **YAML**

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}

/** @Entity */
class Group
{
    // ...
    /**
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

    public function __construct() {
        $this->users = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

The MySQL schema is exactly the same as for the Many-To-Many uni-directional case above.

### 5.9.1. Owning and Inverse Side on a ManyToMany association

For Many-To-Many associations you can chose which entity is the owning and which the inverse side. There is a very simple semantic rule to decide which side is more suitable to be the owning side from a developers perspective. You only have to ask yourself, which entity is responsible for the connection management and pick that as the owning side.

Take an example of two entities `Article` and `Tag`. Whenever you want to connect an Article to a Tag and vice-versa, it is mostly the Article that is responsible for this relation. Whenever you add a new article, you want to connect it with existing or new tags. Your create Article form will probably support this notion and allow to specify the tags directly. This is why you should pick the Article as owning side, as it makes the code more understandable:

```
<?php
class Article
{
    private $tags;

    public function addTag(Tag $tag)
    {
        $tag->addArticle($this); // synchronously updating inverse side
        $this->tags[] = $tag;
    }
}

class Tag
{
    private $articles;

    public function addArticle(Article $article)
    {
        $this->articles[] = $article;
    }
}
```

This allows to group the tag adding on the `Article` side of the association:

```
<?php
$article = new Article();
$article->addTag($tagA);
$article->addTag($tagB);
```

## 5.10. Many-To-Many, Self-referencing

You can even have a self-referencing many-to-many association. A common scenario is where a `User` has friends and the target entity of that relationship is a `User` so it is self referencing. In this example it is bidirectional so `User` has a field named `$friendsWithMe` and `$myFriends`.

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToMany(targetEntity="User", mappedBy="myFriends")
     */
    private $friendsWithMe;

    /**
     * @ManyToMany(targetEntity="User", inversedBy="friendsWithMe")
     * @JoinTable(name="friends",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="friend_user_id", referencedColumnName="id")}
     *      )
     */
    private $myFriends;

    public function __construct() {
        $this->friendsWithMe = new \Doctrine\Common\Collections\ArrayCollection();
        $this->myFriends = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

Generated MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE friends (
    user_id INT NOT NULL,
    friend_user_id INT NOT NULL,
    PRIMARY KEY(user_id, friend_user_id)
) ENGINE = InnoDB;
ALTER TABLE friends ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE friends ADD FOREIGN KEY (friend_user_id) REFERENCES User(id);
```

## 5.11. Mapping Defaults

The `@JoinColumn` and `@JoinTable` definitions are usually optional and have sensible default values. The defaults for a join column in a one-to-one/many-to-one association is as follows:

```
name: "<fieldname>_id"
referencedColumnName: "id"
```

As an example, consider this mapping:

**PHP** **XML** **YAML**

```
<?php
/** @OneToOne(targetEntity="Shipping") */
private $shipping;
```

This is essentially the same as the following, more verbose, mapping:

**PHP** **XML** **YAML**

```
<?php
/**
 * @OneToOne(targetEntity="Shipping")
 * @JoinColumn(name="shipping_id", referencedColumnName="id")
 */
private $shipping;
```

The @JoinTable definition used for many-to-many mappings has similar defaults. As an example, consider this mapping:

**PHP** **XML** **YAML**

```
<?php
class User
{
    //...
    /** @ManyToMany(targetEntity="Group") */
    private $groups;
    //...
}
```

This is essentially the same as the following, more verbose, mapping:

**PHP** **XML** **YAML**

```
<?php
class User
{
    //...
    /**
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="User_Group",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="Group_id", referencedColumnName="id")}
     *      )
     */
    private $groups;
    //...
}
```

In that case, the name of the join table defaults to a combination of the simple, unqualified class names of the participating classes, separated by an underscore character. The names of the join columns default to the simple, unqualified class name of the targeted class followed by "_id". The referencedColumnName always defaults to "id", just as in one-to-one or many-to-one mappings.

If you accept these defaults, you can reduce the mapping code to a minimum.

## 5.12. Collections

Unfortunately, PHP arrays, while being great for many things, are missing features that make them suitable for lazy loading in the context of an ORM. This is why in all the examples of many-valued associations in this manual we will make use of a `Collection` interface and its default implementation `ArrayCollection` that are both defined in the `Doctrine\Common\Collections` namespace. A collection implements the PHP interfaces `ArrayAccess`, `Traversable` and `Countable`.

> The Collection interface and ArrayCollection class, like everything else in the Doctrine namespace, are neither part of the ORM, nor the DBAL, it is a plain PHP class that has no outside dependencies apart from dependencies on PHP itself (and the SPL). Therefore using this class in your model and elsewhere does not introduce a coupling to the ORM.

## 5.13. Initializing Collections

You should always initialize the collections of your `@OneToMany` and `@ManyToMany` associations in the constructor of your entities:

```
<?php
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class User
{
    /** @ManyToMany(targetEntity="Group") */
    private $groups;

    public function __construct()
    {
        $this->groups = new ArrayCollection();
    }

    public function getGroups()
    {
        return $this->groups;
    }
}
```

The following code will then work even if the Entity hasn't been associated with an EntityManager yet:

```
<?php
$group = new Group();
$user = new User();
$user->getGroups()->add($group);
```