CHAPTER 6

References Between Entities

Entities usually are part of a bigger, interconnected *object graph*. They have references to other entities. A `User` holds references to its `Posts`, while a `Post` references a `Category` which references back to the `User`, who (in turn) sets up the `Category`, and so forth. As we will see, there are many different ways to establish connections between entities. Connections are characterized by the *number of items* they connect and the *association's direction*. Let's take a look!

Domain model of the demo app Talking The following code samples are related to the domain model of the demo app introduced earlier in the book. The drawing of the domain model might be helpful as a reference throughout this chapter.

One-to-One Relationship, Unidirectional

In contrast to foreign key relationships in a database, which always join two tables in both directions, this is not true for objects. Therefore, Doctrine 2 differentiates between *unidirectional* and *bidirectional* associations between objects. Unidirectional means that one objects points to the other, but the latter does not have a pointer back.

To set up this type of a relationship between two entities, we simply need to pick a member variable acting as the pointer:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="users")
7  */
8  class User
9  {
10             // [..]
11
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_6

```
12              /**
13              * @OneToOne(targetEntity="Entity\ContactData")
14              */
15              private $contactData;
16
17              // [..]
18      }
```

The `@OneToOne` annotation defines the type of the relationship: a one-to-one relationship. The `targetEntity` attribute defines the entity class to which the pointer points. The class given with the `targetEntity` attribute needs to be fully qualified, including a namespace if applicable. In any case, you must not add a leading backslash.

It's now already possible to reach a referenced entity (`ContactData`) through a loaded `User`, if a getter method has been added:

```
1   <?php
2   var_dump($user->getContactData());
```

However, the simple persistence configuration shown above only works because `ContactData` has a member variable called `id`:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="contact_data")
7   */
8   class ContactData
9   {
10              /**
11              * @Id @Column(type="integer")
12              * @GeneratedValue
13              */
14              private $id;
15
16              // [..]
17  }
```

If the member variable has a different name, such as `contactDataId` or something similar, we need to tell Doctrine 2 about it:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="users")
```

```
 7  */
 8  class User
 9  {
10          // [..]
11
12          /**
13          * @OneToOne(targetEntity="Entity\ContactData")
14          * @JoinColumn(name="id", referencedColumnName="contactDataId")
15          */
16          private $contactData;
17
18          // [..]
19  }
```

Later in this book we will learn more about t the `@JoinColumn` annotation.

Loading a `User` and its `ContactData` as shown above works only if the data structure in the database has also been set up beforehand, in addition to the PHP class and the persistence configuration. If not, all this won't work and errors are reported by Doctrine 2. Based on this situation, one of the following will be true:

- We have an existing data structure given in the database and we adapt the persistence configuration to it.
- We do not have an existing data structure and the persistence configuration has no external restrictions. In this case, the Doctrine 2 *schema tool* can create the data structure in the database based simply on the persistence configuration. More about how to auto-create a data structure in a database can be found in Chapter 9, "Command Line Tools" later in this book.

In any case, it's important that the data structure and the Doctrine 2 persistence mappings match. If not, we will be in trouble for sure. The schema tool can also be used to verify that data structure and persistence mappings match.

In the following, let's assume that we always create the data structure for the demo application using the schema tool. In this case, the `users` table will be created after invoking the proper schema tool commands:

Col	Type	Length	NULL?	KEY	Extras
Id	INT	11		PRI	auto_increment
first_name	VARCHAR	255	Y		
last_name	VARCHAR	255	Y		
gender	INT	11	Y		
name_prefix	VARCHAR	255	Y		
contactData_id	INT	11	Y	UNI	

The `contact_data` table created looks like this:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
email	VARCHAR	255	Y		
phone	VARCHAR	255	Y		

One-to-One Relationship, Bidirectional

In contrast to the unidirectional one-to-one relationship, in a bidirectional relationship two pointers exist: one pointing from object A to object B, and another pointing from object B to object A. It is important that we talk about two separate pointers here, as we will see in a minute. Let's take a `User` entity again with a reference to a `UserInfo` entity:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="users")
*/
class User
{
            // [..]

            /**
            * @OneToOne(targetEntity="Entity\UserInfo")
            */
            private $userInfo;

            // [..]
}
```

If we now want the `UserInfo` entity to point back to the `User` entity, we need to extend the configuration of the `User` entity by adding `inversedBy="user"`:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="users")
*/
class User
{
```

```
10              // [..]
11
12              /**
13              * @OneToOne(targetEntity="Entity\UserInfo", inversedBy="user")
14              */
15              private $userInfo;
16
17              // [..]
18  }
```

We configured the `UserInfo` with a pointer called `$user` to a `UserInfo` entity. The `UserInfo` entity itself looks like this:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="user_info")
7   */
8   class UserInfo
9   {
10              /**
11              * @Id @Column(type="integer")
12              * @GeneratedValue
13              */
14              private $id;
15
16              /** @Column(type="datetime", nullable=true) */
17              private $signUpDate;
18
19              /** @Column(type="datetime", nullable=true) */
20              private $signOffDate = null;
21
22              /**
23              * @OneToOne(targetEntity="Entity\User",  mappedBy="userInfo")
24              */
25              private $user;
26
27              public function setId($id)
28              {
29                      $this->id = $id;
30              }
31
32              public function getId()
33              {
34                      return $this->id;
35              }
```

```
36
37              public function setSignOffDate($signOffDate)
38              {
39                      $this->signOffDate = $signOffDate;
40              }
41
42              public function getSignOffDate()
43              {
44                       return $this->signOffDate;
45              }
46
47              public function setSignUpDate($signUpDate)
48              {
49                      $this->signUpDate = $signUpDate;
50              }
51
52              public function getSignUpDate()
53              {
54                       return $this->signUpDate;
55              }
56
57              public function setUser($user)
58              {
59                       $this->user = $user;
60              }
61
62              public function getUser()
63              {
64                       return $this->user;
65              }
66   }
```

In the `UserInfo` entity we define the target entity class via `targetEntity="Entity\User"` and let Doctrine 2 know by adding `mappedBy="contactData"` that the `User` entity references back via `contactData`.

So far, so good! We just established a bidirectional connection between two entities. Let's now talk about the `inversedBy` and `mappedBy` configuration.

For a moment, let's image there exists a `User` entity and a `UserInfo` entity, belonging together and pointing to each other in a bidirectional manner: `$userInfo` points to a `UserInfo` instance and `$user` points to a `User` instance. If we now want to disconnect the two, we remove both pointers:

```
1   <?php
2   // [..]
3   $user = $em->getRepository('Entity\User')->findOneById($id);
4   $user->getUserInfo()->setUser(null);
5   $user->setUserInfo(null);
6   $em->flush();
```

But what if we remove only one of the two pointers? We would create an inconsistency. Which reference tells Doctrine 2 the truth about the two objects and their association? One entity says "yes, we are connected," the other says "no, we aren't connected at all." Remember that this problem exists only in the object oriented world, not in the relational database universe. In a relational database, references are always bidirectional. There is no concept of a unidirectional relationship. Back to the inconsistency issue: to which entity should Doctrine 2 listen when taking care of persistence? The solution looks like this: when in doubt, Doctrine 2 listens to the entity which carries the `inversedBy` attribute. This means that the following code would not remove the association between the two entities:

```
1   <?php
2   // [..]
3   $user = $em->getRepository('Entity\User')->findOneById($id);
4   $user->getUserInfo()->setUser(null);
5   $em->flush();
```

In contrast, the following code would remove the association:

```
1   <?php
2   // [..]
3   $user = $em->getRepository('Entity\User')->findOneById($id);
4   $user->setUserInfo(null);
5   $em->flush();
```

The reason is that the `User` entity has the `inversedBy` attribute and acts as the so-called *owning side* of the connection. The owning side is the side Doctrine 2 checks to determine whether a connection exists. The other side, the so-called *inverse side*, doesn't matter here. Doctrine 2 doesn't care what the inverse side says.

However, even if we correctly cut the connection between the two from the owning side, this change is durable only after flushing:

```
1   <?php
2   use Doctrine\ORM\EntityManager;
3   // [..]
4   $em = EntityManager::create($dbParams, $config);
5   $em->flush();
```

Until flushing happens, we still have an inconsistency in the running program:

```
1   <?php
2   $user = $em->getRepository('Entity\User')->findOneById($id);
3   $userInfo = $user->getUserInfo();
4   $user->setUserInfo(null);
5   var_dump($user->getUserInfo()); // NULL
6   var_dump($userInfo->getUser()); // object(Entity\User)
7   $em->flush();
8   var_dump($user->getUserInfo()); // NULL
9   var_dump($userInfo->getUser()); // NULL
```

Only once `$em->flush()` has been executed will `var_dump($userInfo->getUser())` finally return `NULL`. Before that, it returns the referenced object. In some cases, this situation may lead to difficult-to-debug issues. A piece of good advice here is to always make sure that both references are removed simultaneously so that the running program stays intact:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="users")
*/
class User
{
            // [..]

            public function removeUserInfo()
            {
                    $this->userInfo->setUser(null);
                    $this->userInfo = null;
            }
}
```

The following code now always deals with a consistent state:

```
<?php
$user = $em->getRepository('Entity\User')->findOneById($id);
$userInfo = $user->getUserInfo();
$user->removeUserInfo();
var_dump($user->getUserInfo()); // NULL
var_dump($userInfo->getUser()); // NULL
$em->flush();
var_dump($user->getUserInfo()); // NULL
var_dump($userInfo->getUser()); // NULL
```

As a good practice, you should always add a persistence supporting method on the owning side.

Let's recap: A bidirectional relationship always has an owning side and an inverse side. Unidirectional relationships have only an owning side, which also does not need to be declared explicitly. Regarding associations, only changes to the owning side are relevant for persistence. Doctrine 2 doesn't care about the inverse side in this concern. The data base table of the entity declared as the owning side holds the foreign key. Which side of the connection is defined as the owning side is up to the application developer.

Given a bidirectional association, the owning side can be identified by spotting the `inversedBy` attribute, while the inverse side carries the `mappedBy` attribute.

One-to-Many Relationship, Bidirectional

In the chapter "Hello, Doctrine 2!" we added a bidirectional one-to-many relationship to the demo app between a `User` and its `Posts`. Let's take a close look at how we got that working. The `User` entity has a member variable called `$posts`:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="users")
7  */
8  class User
9  {
10             /**
11             * @OneToMany(targetEntity="Entity\Post", mappedBy="user")
12             */
13             private $posts;
14
15             // [..]
16 }
```

The `@OneToMany` annotation defines the one-to-many relationship to the `Post` entity (`targetEntity`). As the `mappedBy` attribute is given, it's a bidirectional relationship. The `Post` entity looks like this:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="posts")
7  */
8  class Post
9  {
10             /**
11             * @ManyToOne(targetEntity="Entity\User",  inversedBy="posts")
12             */
13             private $user;
14
15             // [..]
16 }
```

Here we find the counterpart annotation `@ManyToOne`. In a one-to-one relationship, the application developer can decide freely which side to declare as the owning side. That is not the case here. The entity carrying the `@ManyToOne` annotation must become the owning side and get the `inversedBy` attribute.

If (as before) we use the schema tool to create the data structure based on the persistence configuration, the following tables are set up:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	auto_increment
first_name	VARCHAR	255	Y		
last_name	VARCHAR	255	Y		
gender	INT	11	Y		
name_prefix	VARCHAR	255	Y		
contactData_id	INT	11	Y	UNI	
userInfo_id	INT	11	Y	UNI	

The `user_info` table looks like this:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
user_id	INT	11		MUL	
title	VARCHAR	255			
content	VARCHAR	255			

As we can see, table `user_info` holds the foreign key, which may occur more than once. When accessing a `User`'s `Posts` from a given `User` entity, Doctrine uses a `Doctrine\ORM\PersistentCollection` to provide the referenced `Post` entities. It extends other classes such as `\Countable`, `\IteratorAggregate` and `\ArrayAccess`, which makes a `Doctrine\ORM\PersistentCollection` very similar to a regular PHP array, meaning it can easily be used, for example, in PHP `foreach` loops.

If the referenced `Posts` haven't been accessed at least once, the member variable `$user` has the value `null` and has not yet been assigned an object of class `Doctrine\ORM\PersistentCollection`. If one wants to work with the collection before it has been set up by Doctrine 2, this could be an issue. Therefore, as a good practice, a member variable holding a persistent association should always be initialized:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="users")
```

```
 7  */
 8  class User
 9  {
10              // [..]
11
12              public function construct()
13              {
14                      $this->posts = new \Doctrine\Common\Collections\
                        ArrayCollection();
15              }
16
17              // [..]
18  }
```

Many-to-Many Relationship, Unidirectional

In our demo app, one `User` may act in different `Role`s with different access rights. Let's distinguish between base- and pro-users and administrators. `User` with the base `Role` can read `Post`s, but can't write `Post`s. Pro-users can also write `Post`s. Administrators can manage the overall application and various settings. We design the relationship so that, from given a `User` entity, we can access the `User`'s `Role`s, but not the other way around. To do so, we simply need to add the following persistence configuration:

```
 1  <?php
 2  namespace Entity;
 3
 4  /**
 5  * @Entity
 6  * @Table(name="users")
 7  */
 8  class User
 9  {
10          // [..]
11
12          /**
13          * @ManyToMany(targetEntity="Entity\Role")
14          **/
15          private $roles;
16
17          // [..]
18  }
```

The `Role` entity definitions are straightforward as well:

```
1  <?php
2  namespace Entity;
3
```

```
4   /**
5   * @Entity
6   * @Table(name="role")
7   */
8   class Role
9   {
10          /**
11          * @Id @Column(type="integer")
12          * @GeneratedValue
13          */
14          private $id;
15
16          /** @Column(type="string") */
17          private $label;
18
19          public function setId($id)
20          {
21                  $this->id = $id;
22          }
23
24          public function getId()
25          {
26                  return $this->id;
27          }
28
29          public function setLabel($label)
30          {
31                  $this->label = $label;
32          }
33
34          public function getLabel()
35          {
36                  return $this->label;
37          }
38  }
```

Running the schema tool to create the data structure in the database, we now have a new table called `role`:

Col	Type	Length	NULL?	KEY	Extras
Id	INT	11		PRI	auto_increment
Label	VARCHAR	255			

Also, we find a table called `user_role` allowing us to persist a many-to-many reference:

Col	Type	Length	NULL?	KEY	Extras
user_id	INT	11		MUL	
role_id	INT	11		MUL	

The reason that the join table shown above looks like it does is that Doctrine 2 has again applied a default configuration to the `@JoinTable` annotation, which itself is not given with our code, but which implicitly is considered by Doctrine 2. If needed, our join table can be defined differently. For example, if an existing data structure in a database needs to be used:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="users")
7  */
8  class User
9  {
10             // [..]
11
12             /**
13             * @ManyToMany(targetEntity="Entity\Role")
14             * @JoinTable(name="users_roles",
15             *      joinColumns={@JoinColumn(name="user",
                       referencedColumnName="id")},
16             *      inverseJoinColumns={@JoinColumn(name="role",
17             *                          referencedColumnName="id")
18             *               }
19             * )
20             **/
21             private $roles;
22
23             // [..]
24 }
```

The persistence configuration now looks much more complex and tells Doctrine to deal with a join table that looks like this:

Col	Type	Length	NULL?	KEY	Extras
user	INT	11		MUL	
role	INT	11		MUL	

The `@JoinTable` annotation has an attribute called `name` to define the join table name as well as the attributes `joinColumns` and `inverseJoinColumns`, both of which use another annotation, called `@JoinColumn`, to map the individual columns.

Many-to-Many Relationship, Bidirectional

The author of a `Post` (a `User`) should be able to assign one or more `Tags`. A `Tag` itself may be reused in different `Posts`. In contrast to the relationship between a `User` and its `Roles`, we design the association to be bidirectional. First, let's set up the new `Tag` entity:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="tag")
*/
class Tag
{
            /**
            * @Id @Column(type="integer")
            * @GeneratedValue
            */
            private $id;

            /** @Column(type="string") */
            private $label;

            /**
            * @ManyToMany(targetEntity="Entity\Post", mappedBy="tags")
            */
            private $posts;

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

            public function setId($id)
            {
                      $this->id = $id;
            }

            public function getId()
            {
                      return $this->id;
            }
```

```
38
39              public function setLabel($label)
40              {
41                        $this->label = $label;
42              }
43
44              public function getLabel()
45              {
46                        return $this->label;
47              }
48
49              public function setPosts($posts)
50              {
51                        $this->posts = $posts;
52              }
53
54              public function getPosts()
55              {
56                        return $this->posts;
57              }
58  }
```

The Post entity needs to be extended, as well, by a new member variable:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="posts")
7   */
8   class Post
9   {
10              // [..]
11
12              /**
13              * @ManyToMany(targetEntity="Entity\Tag",  inversedBy="posts")
14              */
15              private $tags;
16
17              public function __construct()
18              {
19                        $this->tags = new \Doctrine\Common\Collections\
                          ArrayCollection();
20              }
21
22              // [..]
23  }
```

When setting up a many-to-many relationship, you can freely choose the owning side.

The schema tool, once more, creates the join table `post_tag` for us, if we run the proper commands:

Col	Type	Length	NULL?	KEY	Extras
post_id	INT	11		MUL	
tag_id	INT	11		MUL	

Again, defaults are applied here. If desired, the `@JoinTable` annotation can be used to alter the labels of the table and its columns.

One-to-Many Relationship, Unidirectional

A `User` can set up multiple `Category` entities for its `Posts`. We will use a unidirectional association for this domain model aspect, which requires—in contrast to the bidirectional one-to-many relationship—a join table as well. Also, we need to use the `@ManyToMany` annotation, even though we don't set up a many-to-many annotation (this might be somewhat confusing). With the help of a unique constraint, it finally results in a one-to-many relationship. The `User` entity is extended by the `$categories` member variable:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="users")
*/
class User
{
        // [..]

        /**
        * @ManyToMany(targetEntity="Entity\Category")
        * @JoinTable(name="users_categories",
        *      joinColumns={@JoinColumn(name="user",
        referencedColumnName="id")},
        *      inverseJoinColumns={@JoinColumn(name="category",
        *                  referencedColumnName="id", unique=true)}
        * )
        */
        private $categories;

        public function __construct()
        {
                // [..]
```

```
25                  $this->categories = new \Doctrine\Common\Collections\
                    ArrayCollection();
26          }
27
28          public function setCategories($categories)
29          {
30                  $this->categories = $categories;
31          }
32
33          public function getCategories()
34          {
35                  return $this->categories;
36          }
37
38          // [..]
39  }
```

The annotations and attributes used are already known from the many-to-many relationship. The `unique=true` configuration, which makes it a one-to-many relationship, is new. The `Category` entity is simple:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="category")
7   */
8   class Category
9   {
10          /**
11          * @Id @Column(type="integer")
12          * @GeneratedValue
13          */
14          private $id;
15
16          /** @Column(type="string") */
17          private $label;
18
19          public function setId($id)
20          {
21                  $this->id = $id;
22          }
23
24          public function getId()
25          {
26                  return $this->id;
27          }
28
```

```
29          public function setLabel($label)
30          {
31                  $this->label = $label;
32          }
33
34          public function getLabel()
35          {
36                  return $this->label;
37          }
38  }
```

Many-to-One Relationship, Unidirectional

Multiple posts can be grouped in a Category, but each Post can be in only one Category. To allow access to a Category from a Post, the Post must be extended:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="posts")
7   */
8   class Post
9   {
10
11          // [..]
12
13          /**
14          * @ManyToOne(targetEntity="Entity\Category")
15          * @JoinColumn(name="category_id", referencedColumnName="id")
16          **/
17          private $category;
18
19          // [..]
20
21          public function setCategory($category)
22          {
23                  $this->category = $category;
24          }
25
26          public function getCategory()
27          {
28                  return $this->category;
29          }
30  }
```

And that's it—no more configuration is needed, as it's a unidirectional association. The `Category` doesn't need to be extended. In fact, even the `@JoinColumn` annotation is redundant, because its configuration is identical with Doctrine's defaults.

One-to-One Relationship, Self-Referencing

Doctrine 2 allows us to define associations between entities of the same type, so-called *self-referencing* relations. In our demo app, a `User` can declare another `User` as its life partner. Both ends of the association allow us to access the referenced life partner. The persistence configuration needed for a self-referencing association looks like this:

```
<?php
namespace Entity;

/**
* @Entity
* @Table(name="users")
*/
class User
{
        /**
        * @OneToOne(targetEntity="Entity\User")
        **/
        private $lifePartner;

        // [..]

        public function setLifePartner($lifePartner)
        {
                $this->lifePartner = $lifePartner;
        }

        public function getLifePartner()
        {
                return $this->lifePartner;
        }
}
```

If the database structure is created by the schema tool, Doctrine 2 automatically adds a column called `lifeParter_id` to the `users` table to maintain the reference. Again, Doctrine's defaults are at play here. If needed, you can add the `@JoinColumn` annotation and overwrite the defaults:

```
<?php
// [..]

/**
```

```
 5   * @OneToOne(targetEntity="Entity\User")
 6   * @JoinColumn(name="partner", referencedColumnName="id")
 7   **/
 8   private $lifePartner;
 9
10   // [..]
```

One-to-Many Relationship, Self-Referencing

With a self-referencing one-to-many relationship, a category tree can be built. A `Category` may have multiple child categories and one parent category (the root node won't have a parent category). The persistence configuration for a self-referencing one-to-many relationship is similar to a regular bidirectional one-to-many relationship:

```
 1   <?php
 2   namespace Entity;
 3
 4   /**
 5   * @Entity
 6   * @Table(name="category")
 7   */
 8   class Category
 9   {
10           /**
11           * @OneToMany(targetEntity="Entity\Category", mappedBy="parent")
12           **/
13           private $children;
14
15           /**
16           * @ManyToOne(targetEntity="Entity\Category",
               inversedBy="children")
17           * @JoinColumn(name="parent_id", referencedColumnName="id")
18           **/
19           private $parent;
20
21           // [..]
22
23           public function __construct() {
24                   $this->children = new \Doctrine\Common\Collections\
                     ArrayCollection();
25           }
26
27           public function setParent($parent)
28           {
29                   $this->parent = $parent;
30           }
31
```

```
32          public function getParent()
33          {
34                  return $this->parent;
35          }
36
37          // [..]
38  }
```

The configuration shown above extends table `category` by column `parent_id`, if the already developed data structure is extended using Doctrine's schema tool. The relationship is designed in a way that allows us to reach a parent and children from a given `Category`.

Many-to-many Relationship, Self-Referencing

Many-to-many self-referencing relationships can be defined as well. In our demo app, this type of a relationship is used to describe the social network of a `User`:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="users")
7   */
8   class User
9   {
10          // [..]
11
12          /**
13          * @ManyToMany(targetEntity="Entity\User")
14          * @JoinTable(name="friends",
15          *      joinColumns={@JoinColumn(name="user_id",
16          *                          referencedColumnName="id")},
17          *      inverseJoinColumns={@JoinColumn(name="friend_user_id",
18          *                      referencedColumnName="id")}
19          * )
20          **/
21          private $myFriends;
22
23          public function __construct()
24          {
25                  // [..]
26                  $this->myFriends = new \Doctrine\Common\Collections\
                    ArrayCollection();
27          }
28
```

```
29              // [..]
30
31              public function setMyFriends($myFriends)
32              {
33                      $this->myFriends = $myFriends;
34              }
35
36              public function getMyFriends()
37              {
38                      return $this->myFriends;
39              }
40      }
```

If the schema tool is applied, it will report an error here, because Doctrine 2 wants to label both columns of the join table with "user_id," which is invalid. In this case, we will need to add the `@JoinTable` annotation to provide a different persistence configuration. Also, by default, the join table will be called `users_users`, if not defined otherwise. In our case, we tell Doctrine 2 to call it `friends`, a more meaningful table name in our case. If you are dealing with an existing data structure, custom configuration would be necessary anyway.

Summary

Another important milestone is reached! We are now able to design and persist complex PHP object graphs connecting multiple individual entities. This is a huge step forward and completes most of the work needed related to configuring persistence using Doctrine. In the next chapter, we will start to create and manipulate entities and their associations programmatically using PHP code.