CHAPTER 4

# Hello, Doctrine 2!

In the previous chapter we learned about the complexity of an ORM the hard way by implementing our own persistence code. Doctrine 2 instead provides full transparent persistence for PHP objects by implementing the so called "Data Mapper Pattern". Essentially, Doctrine allows the programmer to focus on the object-oriented business logic and takes the pain out of PHP object persistence.

**Don't worry!** We run through Doctrine 2 quickly in this chapter, but we will look into each individual aspect of it in depth later in the book.

## Installation

The easiest way to install Doctrine 2 is by using "Composer," which can be downloaded from its website.[1] Store the PHP archive (phar) file in the root directory of the application. Then, create a file called `composer.json` with the following contents (you may need to adjust the version number given):

```
{
     "require": {
            "doctrine/orm": "2.3.1"
      }
}
```

In the root directory of the application, execute the command

```
$ php composer.phar install
```

to download Doctrine 2 to the `vendor` subfolder. Composer makes it available to the application by configuring autoloading for its classes.

[1] http://getcomposer.org/download/

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_4

If you receive a composer error or warning, you may want to update composer itself to its latest version first by running

```
$ php composer.phar self-update
```

You then only need to add

```php
<?php
// [..]
if (file_exists('vendor/autoload.php')) {
    $loader = include 'vendor/autoload.php';
}
```

to the `index.php` file. Once downloaded, all Doctrine 2 classes are loaded automatically on demand. Nice!

## A First Entity

Based on the first section of this book, in which we developed our own little ORM system, we now can radically simplify the code needed by adding Doctrine 2 annotations to the `User` entity:

```php
<?php
namespace Entity;

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

        /** @Column(type="string", name="first_name", nullable=true) */
        private $firstName;

        /** @Column(type="string", name="last_name", nullable=true) */
        private $lastName;

        /** @Column(type="string", nullable=true) */
        private $gender;

```

```
    /** @Column(type="string", name="name_prefix", nullable=true)  */
    private $namePrefix;

    const GENDER_MALE = 0;
    const GENDER_FEMALE = 1;

    const GENDER_MALE_DISPLAY_VALUE = "Mr.";
    const GENDER_FEMALE_DISPLAY_VALUE = "Ms.";

    public function assembleDisplayName()
    {
        $displayName = '';

        if ($this->gender == self::GENDER_MALE) {
            $displayName .= self::GENDER_MALE_DISPLAY_VALUE;
        } elseif ($this->gender == self::GENDER_FEMALE) {
            $displayName .= self::GENDER_FEMALE_DISPLAY_VALUE;
        }

        if ($this->namePrefix) {
            $displayName .= ' ' . $this->namePrefix;
        }

        $displayName .= ' ' . $this->firstName . ' ' . $this->lastName;

        return $displayName;
    }

    public function setFirstName($firstName)
    {
        $this->firstName = $firstName;
    }

    public function getFirstName()
    {
        return $this->firstName;
    }

    public function setGender($gender)
    {
        $this->gender = $gender;
    }

    public function getGender()
    {
        return $this->gender;
    }
```

```

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

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

        public function setLastName($lastName)
        {
                $this->lastName = $lastName;
        }

        public function getLastName()
        {
                return $this->lastName;
        }

        public function setNamePrefix($namePrefix)
        {
                $this->namePrefix = $namePrefix;
        }

        public function getNamePrefix()
        {
                return $this->namePrefix;
        }
}
```

The code in index.php now can be changed to the following lines of code to read and modify entities using Doctrine 2:

```
<?php
include '../entity/User.php';
include '../vendor/autoload.php';

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$paths = array(__DIR__ . "/../entity/");
$isDevMode = true;

$dbParams = array(
        'driver' => 'pdo_mysql',
        'user' => 'root',
        'password' => '',
        'dbname' => 'app',
);

$config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
$em = EntityManager::create($dbParams, $config);
$user = $em->getRepository('Entity\User')->findOneById(1);
echo $user->assembleDisplayName() . '<br />';
$user->setFirstname('Moritz');
$em->persist($user);
$em->flush();
```

Please note that we replaced our custom entity manager with the one provided by Doctrine as well as the user repository with it findOneById method. All this code is available out-of-the-box. The only custom code still needed is the user entity definition.

That's it! Easy!

## A First Association

Dealing with the association is easy as well. The User entity has to be changed to:

```
<?php
namespace Entity;

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

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

        // [..]

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

        // [..]
}
```

Now add some more annotations to the `Post` entity:

```
<?php
namespace Entity;

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

        /**
        * @ManyToOne(targetEntity="Entity\User", inversedBy="posts")
        * @JoinColumn(name="user_id", referencedColumnName="id")
        */
        private $user;

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

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

        public function setUserId($user_id)
        {
                $this->user_id = $user_id;
        }

        public function getUserId()
        {
                return $this->user_id;
        }

        public function setContent($content)
        {
                $this->content = $content;
        }

        public function getContent()
        {
                return $this->content;
        }

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

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

        public function setTitle($title)
        {
                $this->title = $title;
        }

        public function getTitle()
        {
                return $this->title;
        }
}
```

Once the User is loaded, you can iterate over the User's posts as before:

```
<?php
// [..]
$user = $em->getRepository('Entity\User')->findOneById(1);
?>
<h1><?echo $user->assembleDisplayName(); ?></h1>
<ul>
        <?php foreach($user->getPosts() as $post) {?>
                <li><?php echo $post->getTitle(); ?></li>
        <?php } ?>
</ul>
```

In fact, with Doctrine 2 included, we don't need most of the code we wrote to implement the exact same functionality by hand:

```
entity/
        Post.php
        User.php
public/
        index.php
```

It couldn't be much easier—thank you, Doctrine 2! Before we add more features to our demo app, we will now look at some basic concepts of Doctrine 2.

## Core Concepts at a Glance

Before we take a deep dive into the details of Doctrine 2, let's step back and look at its core concepts.

We already defined an *entity* as an *object* with *identity* that is managed by Doctrine 2 and persisted in a database. Entities are the domain objects of an application. Saving entities and retrieving them from the database is essential to an application. The term "entity" is used with two different meanings throughout this book: on one hand, it is used for single, persistent object; on the other hand, it is used for PHP classes that act as templates for persistent objects. With Doctrine 2, a persistent object, an entity, is always in a specific state. This state can be "NEW," "MANAGED," "DETACHED," or "REMOVED." We will learn about this later in the book.

The entity manager is Doctrine's core object. As an application developer, one does interact a lot with the entity manager. The entity manager is responsible for persisting new entities, and also takes care of updating and deleting them. Repositories are retrieved via the entity manager.

Simply speaking, a *repository* is a container for all entities of a specific type. It allows you to look-up entities via so-called *finder methods* or *finders*. Some finders are available by default, others may be added manually if needed.

Doctrine's *database access layer (DBAL)* is the foundation for the Doctrine 2 ORM library. It's a stand-alone component that can be used even without using Doctrine's ORM capabilities. Technically, DBAL is a wrapper for PHP's PHP data objects (PDO) extension that makes dealing with databases even easier compared to PHP alone. And again, Doctrine 2 ORM makes dealing with databases even easier than with DBAL alone.

## Summary

What a ride! By simply pulling in Doctrine as a persistence library into our application, we could already reduce the complexity of our application code dramatically. It's now time to explorer all the powerful features of Doctrine in more detail!