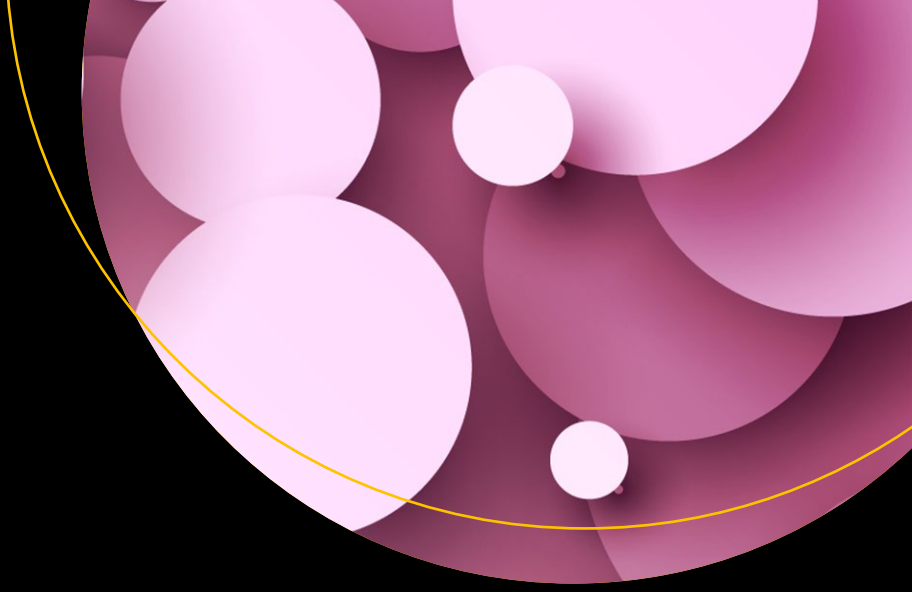

PHP Persistence

Concepts, Techniques and Practical Solutions with Doctrine

Michael Romer

Apress®

PHP Persistence

Concepts, Techniques and Practical Solutions with Doctrine

Michael Romer

Apress®

PHP Persistence: Concepts, Techniques and Practical Solutions with Doctrine

Michael Romer
Munster, Nordrhein-Westfalen
Germany

ISBN-13 (pbk): 978-1-4842-2558-5 ISBN-13 (electronic): 978-1-4842-2559-2
DOI 10.1007/978-1-4842-2559-2

Library of Congress Control Number: 2016961535

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Deepak Vohra
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Alexander Krider
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Michael Romer holds a degree in computer science from Westfälische Hochschule, Germany. He manages agile software development projects and teams using Scrum and Kanban. Michael is a Certified Scrum Professional, Certified Scrum Master and Certified Scrum Product Owner. He's worked for eBay and today helps web startups with their products and technology. He also codes web applications, mostly with PHP and Doctrine. Michael heads the product and business development department of a German publishing house.

About the Technical Reviewer

Deepak Vohra is a consultant and a principal member of the NuBean.com software company. Deepak is a Sun-certified Java programmer and Web component developer. He has worked in the fields of XML, Java programming, and Java EE for over seven years. Deepak is the coauthor of *Pro XML Development with Java Technology* (Apress, 2006). He is also the author of the *JDBC 4.0* and *Oracle JDeveloper for J2EE Development, Processing XML Documents with Oracle JDeveloper 11g, EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*, and *Java EE Development in Eclipse IDE* (Packt Publishing). He has served as the technical reviewer on *WebLogic: The Definitive Guide (O'Reilly Media, 2004)* and *Ruby Programming for the Absolute Beginner* (Cengage Learning PTR, 2007).

CHAPTER 1

About This Book

"I would have written a shorter book, but I did not have the time." Freely adapted from Blaise Pascal (French mathematician, logician, physicist, and theologian).

Software Version

This book is based on Doctrine 2.3; however, most of its content will still be valid for later versions.

Database System

All examples in this book are based on MySQL.[1] However, Doctrine 2 supports other database management systems (DBMS) such as PostgreSQL[2] and SQLite.[3] Most examples will work with other DBMS right away; some may need to be adapted.

[1] http://www.mysql.com
[2] http://www.postgresql.org/
[3] http://www.sqlite.org/

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2559-2_1) contains supplementary material, which is available to authorized users.

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_1

Code Downloads

Most of the code shown in this book can be found in a public repository on GitHub.[4] While the code of Chapter 3 ("A Self-Made ORM") can be found on the "master" branch,[5] there is another branch available, called "doctrine,"[6] which holds the modifications made in Chapter 4 ("Hello, Doctrine 2!"). There is a third branch called "application"[7] holding the modifications made in the later chapters. As the demo application grows throughout the book, the `application` branch also includes a basic model-view-controller (MVC) framework, called Slim,[8] to better structure the application's code, as well as Twitter's Bootstrap[9] for some basic UI styling.

Conventions Used in This Book

Code listings are highlighted and have line numbering. Listings that start with a $ symbol represent the command line, lines stating with a ➤ symbol represent command output. New terms are written in *italic* on first usage.

[4] https://github.com/michael-romer/doctrine2buch
[5] https://github.com/michael-romer/doctrine2buch
[6] https://github.com/michael-romer/doctrine2buch/tree/doctrine
[7] https://github.com/michael-romer/doctrine2buch/tree/application
[8] http://www.slimframework.com/
[9] http://getbootstrap.com

CHAPTER 2

Introduction

Object-Oriented Programming (OOP) and the Domain Model

PHP developers nowadays generally think and code in an object-oriented way. Application functionality is built using classes, objects, methods, inheritance, and other object-oriented techniques. In the beginning, OOP was used primarily for the general technical aspects of applications, such as MVC frameworks or logging and mailing libraries. All these components can be used more or less "as is" in other applications, regardless of the domains those applications inhabit: for example, e-commerce, portal, or community site. For complex systems, or if aspects such as maintainability and extensibility are important, OOP is also an advantage in the domain-specific code. Basically, every application consists of two types of code: general technical code and domain-specific code. General technical code is often reusable when built as a library or framework; domain-specific code is often too customized to be reused.

Object-oriented domain-specific code is characterized by the existence of a so-called *domain model*. A domain model includes:

- Classes and objects representing the main concepts of a domain, the so-called *entities*. These elements can also be *value objects*, to be precise, but compared to entities, value objects don't have a persistent identity. In an online shop, the main elements would be "Customer," "Order," "Product," "Cart," and so forth.
- Associations between domain-specific classes and objects. In our online shop example, an `Order` would have at least one `Customer` that it references as well as one or more references to the `Product(s)` ordered.
- Domain-specific functions implemented as a part of an entity. In our online shop, a `Cart` could have a `calculateTotalPrice()` method to calculate the final price based on the items in the `Cart` and their quantities.

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_2

- Functions that span multiple entities usually implemented in so-called *services*, simply because they cannot clearly be assigned to one single entity. In our online shop, the "Checkout" service may take care of lowering the inventory, invoicing, updating the order history, etc. The service deals with several entities at once.
- Domain-specific objects used instead of generic data containers such as PHP arrays whenever possible (exceptions prove the rule here).
- Business logic (such as business rules) implemented within the domain objects of an application whenever possible, not in controllers, for example (controllers are used in MVC–based frameworks to handle user input).

The main advantage of the domain model lies in having the domain-specific code centrally defined in classes and objects, an approach which facilitates maintenance and changes. The possibility of incidentally breaking a function, when changing or extending code, drops. By isolating domain-specific code from general technical code, portability is supported. That's helpful, for example, when migrating from one application framework to another.

Besides the advantages mentioned above, the domain model also supports teamwork. Often, when building and launching a new software product, programmers work together with business and marketing folks. A domain model can bridge the mental gap between business and IT by unifying the terminology used. This alone makes a domain model invaluable.

Demo Application

Concrete examples make things easier to understand. Throughout this book, a demo application called "Talking" will help to put theory into practice. `Talking` is a (simple) web application allowing users to publish content online. Figure 2-1 shows the application's domain model:

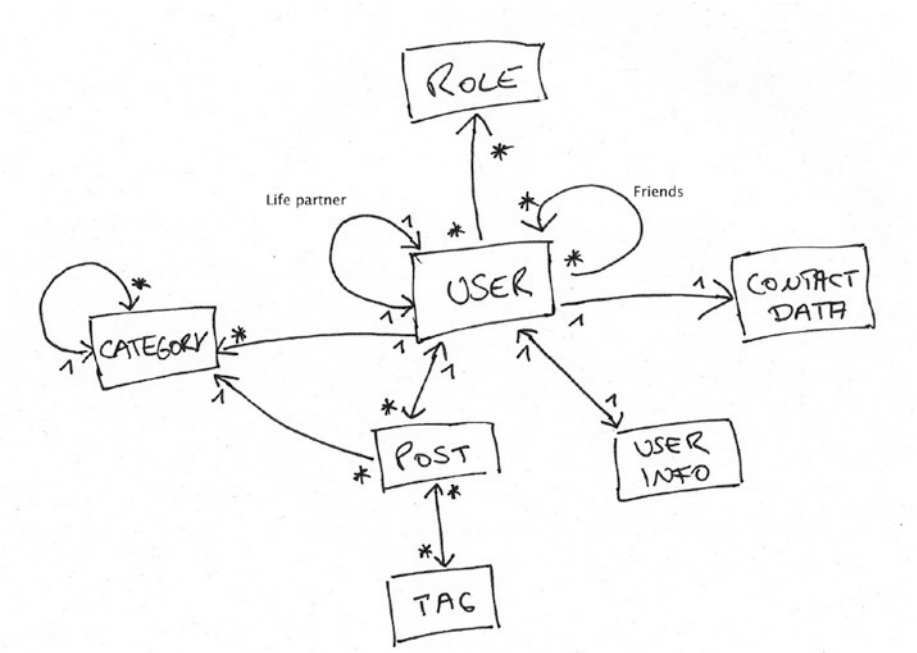

Figure 2-1. *Demo application "Talking" - Domain Model*

In the `Talking` demo application, a `User` can write `Posts`. A `Post` always has only one author (the `User`), and the two entities reference each other. A `User` can act in one or more `Roles`. A `User` references its `Roles`, but from a given `Role`, one cannot access the `Users` who reference the Role. A `User` references `UserInfo` holding the date of registration and the date of de-registration, if available. `UserInfo` references back to a `User`. A `User` references `ContactData` where the `User's` email and phone number are saved. `ContactData` does not reference back to its `User`. A `User` may reference another `User` as its life partner. The `User`'s life partner references back if known. A `User` may have an unlimited number of friends. Given a `User`, one can identify its friends, but there is no reference back. The `Post` of a `User` can have an unlimited number of `Tags`. A `Tag` can be reused in several `Posts`. There is a bidirectional association between a `Post` and its `Tags`. A `Post` references its `Category`, however, there is no reference from a `Category` to its `Posts`. A `Category` can have subcategories which it references, as well as a parent `Category`, if given. Categories are user-specific. A `User` references its categories, but there is no reference back.

Our demo domain model is designed to use as many different association types as possible, so that we can see most of the features of Doctrine 2 in action in the scope of the demo application. In a real application, the domain model would probably look a bit different. As you can see, not every association between entities is bidirectional. And as we will see later in this book, this is an essential feature of an object-relational mapping (ORM) system such as Doctrine 2 living in the object-oriented world. In relational databases, there are no unidirectional associations; they are always bidirectional by design.

CHAPTER 3

A Self-Made ORM

I believe that showing is always better than telling. Therefore, instead of simply listing the pros and cons of Doctrine 2, I like to demonstrate what it looks like not having it in your PHP toolkit when dealing with PHP objects and relational databases.

Jump straight into Doctrine 2? This section illustrates why Doctrine 2 is such a big help for the application developer. Step by step, we will implement, on our own, certain ORM features that are already included in Doctrine 2. This chapter is not needed to learn Doctrine 2, but it helps in understanding why one should learn it.

Loading an Entity

A domain model is a good thing. As long as the application developer acts in the familiar object-oriented world, there are no obstacles to designing and implementing a domain model. Design and implementation become harder, though, if it becomes necessary to persist objects of the domain model in relational databases or to load and reconstruct previously stored objects. There is, for instance, the fact that objects are more than just dumb data structures. With their methods, they have behavior as well. Let's consider the `User` entity definition of the `Talking` demo application:

```
 1  <?php
 2  namespace Entity;
 3
 4  class User
 5  {
 6      private $id;
 7      private $firstName;
 8      private $lastName;
 9      private $gender;
10      private $namePrefix;
11
12      const GENDER_MALE = 0;
13      const GENDER_FEMALE = 1;
14
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_3

```
15      const GENDER_MALE_DISPLAY_VALUE = "Mr.";
16      const GENDER_FEMALE_DISPLAY_VALUE = "Ms.";
17
18      public function assembleDisplayName()
19      {
20           $displayName = '';
21
22           if ($this->gender == self::GENDER_MALE) {
23               $displayName .= self::GENDER_MALE_DISPLAY_VALUE;
24           } else if ($this->gender == self::GENDER_FEMALE) {
25               $displayName .= self::GENDER_FEMALE_DISPLAY_VALUE;
26           }
27
28           if ($this->namePrefix) {
29               $displayName .= ' ' . $this->namePrefix;
30           }
31
32           $displayName .= ' ' . $this->firstName . ' ' . $this->lastName;
33
34           return $displayName;
34      }
36
37      public function setFirstName($firstName)
38      {
39          $this->firstName = $firstName;
40      }
41
42      public function getFirstName()
43      {
44           return $this->firstName;
45      }
46
47      public function setGender($gender)
48      {
49          $this->gender = $gender;
50      }
51
52      public function getGender()
53      {
54          return $this->gender;
55      }
56
57      public function setId($id)
58      {
59          $this->id = $id;
60      }
61
62      public function getId()
```

```
63      {
64          return $this->id;
65      }
66
67      public function setLastName($lastName)
68      {
69          $this->lastName = $lastName;
70      }
71
72      public function getLastName()
73      {
74          return $this->lastName;
75      }
76
77      public function setNamePrefix($namePrefix)
78      {
79          $this->namePrefix = $namePrefix;
80      }
81
82      public function getNamePrefix()
83      {
84          return $this->namePrefix;
85      }
86  }
```

Listing 1.1

Interesting here is the method `assembleDisplayName()`. It creates the "display name" for a user based on a user's data. The display name is used to print a post's author.

```
1   <?php
2   include('../entity/User.php');
3
4   $user = new Entity\User();
5   $user->setFirstName('Max');
6   $user->setLastName('Mustermann');
7   $user->setGender(0);
8   $user->setNamePrefix('Prof. Dr');
9
10  echo $user->assembleDisplayName();
```

Listing 1.2

The code in Listing 1.2 results in:

```
1   Mr. Prof. Dr. Max Mustermann
```

The method `assembleDisplayName()` therefore defines a specific behavior of a `User` object.

If a user's master data is retrieved from the database, it must be transformed in a way that allows the behavior described above to be attached to it. In other words, the user's master data, retrieved from the database, must be transformed into a `User` object. Within the application, we always want to deal with objects of our domain so that we can easily create a `User's` display name by simply calling its `assembleDisplayName()` method. Let's build that into our ORM tool.

First, we set up the database structure:

```
1   CREATE TABLE users(
2            id int(10) NOT NULL auto_increment,
3            first_name varchar(50) NOT NULL,
4            last_name varchar(50) NOT NULL,
5            gender ENUM('0','1') NOT NULL,
6            name_prefix varchar(50) NOT NULL,
7            PRIMARY KEY (id)
8   );
```

Then, let's add dummy data for "Max Mustermann" (the German "John Doe," by the way):

```
1   INSERT INTO users (first_name, last_name, gender, name_prefix)
2   VALUES('Max', 'Mustermann', '0', 'Prof. Dr.');
```

Now, if we want to reconstruct the `User` object from the database, we can do it like this:

```
 1   <?php
 2   include('../entity/User.php');
 3
 4   $db = new \PDO('mysql:host=localhost;dbname=app', 'root', '');
 5   $userData = $db->query('SELECT * FROM users WHERE id = 1')->fetch();
 6
 7   $user = new Entity\User();
 8   $user->setId($userData['id']);
 9   $user->setFirstName($userData['first_name']);
10   $user->setLastName($userData['last_name']);
11   $user->setGender($userData['gender']);
12   $user->setNamePrefix($userData['name_prefix']);
13
14   echo $user->assembleDisplayName();
```

Database credentials in live systems In a live system, you use strong keywords and don't work with user "root," right? The code shown above is just an example …

With these lines of code, we started implementing our own ORM system, which allows reconstructing domain objects by fetching data from a database. Let's further improve it.

To encapsulate the "data mapping" shown above, we move the code into its own class:

```
<?php
namespace Mapper;

class User
{
        private $mapping = array(
                'id' => 'id',
                'firstName' => 'first_name',
                'lastName' => 'last_name',
                'gender' => 'gender',
                'namePrefix' => 'name_prefix'
        );

        public function populate($data, $user)
        {
                $mappingsFlipped = array_flip($this->mapping);

                foreach($data as $key => $value) {
                        if(isset($mappingsFlipped[$key])) {
                                call_user_func_array(
                                        array($user, 'set'. ucfirst
                                        ($mappingsFlipped[$key])),
                                        array($value)
                                );
                        }
                }

                return $user;
        }
}
```

The User mapper is not perfect, but it does the job. Now, our invoking code looks like this:

```
<?php
include_once('../entity/User.php');
include_once('../mapper/User.php');

$db = new \PDO('mysql:host=localhost;dbname=app', 'root', '');
$userData = $db->query('SELECT * FROM users WHERE id = 1')->fetch();

$user = new Entity\User();
$userMapper = new Mapper\User();
$user = $userMapper->populate($userData, $user);

echo $user->assembleDisplayName();
```

However, we can make the mapping process even easier by moving the SQL statement into its own object, a so-called *repository*:

```
<?php
namespace Repository;

include_once('../entity/User.php');
include_once('../mapper/User.php');

use Mapper\User as UserMapper;
use Entity\User as UserEntity;

class User
{
        private $em;
        private $mapper;

        public function __construct($em)
        {
                $this->mapper = new UserMapper;
                $this->em = $em;
        }

        public function findOneById($id)
        {
                $userData = $this->em
                        ->query('SELECT * FROM users WHERE id = ' . $id)
                         ->fetch();

                return $this->mapper->populate($userData, new
                UserEntity());
        }
}
```

Lastly, we move the code that connects to the database into a class called `EntityManager` and make the new `User` repository available through it:

```
<?php

include_once('../repository/User.php');

use Repository\User as UserRepository;

class EntityManager
{
        private $host;
        private $db;
        private $user;
```

```
12          private $pwd;
13          private $connection;
14          private $userRepository;
15
16          public function construct($host, $db, $user, $pwd)
17          {
18                  $this->host = $host;
19                  $this->user = $user;
20                  $this->pwd = $pwd;
21
22                  $this->connection = new \PDO(
23                          "mysql:host=$host;dbname=$db",
24                          $user,
25                          $pwd);
26
27                  $this->userRepository = null;
28          }
29
30          public function query($stmt)
31          {
32                   return $this->connection->query($stmt);
30          }
34
35          public function getUserRepository()
36          {
37                  if (!is_null($this->userRepository)) {
38                          return $this->userRepository;
39                  } else {
40                          $this->userRepository = new UserRepository($this);
41                          return $this->userRepository;
42                  }
43          }
44  }
```

The `EntityManager` now acts as the main entry point; it opens the database connection as well making database queries available to client code. After this refactoring, the result remains the same:

```
1   Mr. Prof. Dr. Max Mustermann
```

We wrote a whole bunch of code, and yet we can't do anything more than read data from a database and make an object out of it. We didn't really push our own application forward. Looks like building an ORM system is hard work and time-consuming. And we've just started. Let's spend some more time enhancing our ORM so that, later, we will appreciate even more what Doctrine 2 can do for us.

Saving an Entity

So far, we have implemented a trivial use case: loading a single object from the database based on a given ID. But what about writing operations? Actually, there are two types of write operations: *inserts* and *updates*. Let's first deal with the insert operation by adding an `extract()` method to the `User` mapper:

```
<?php
namespace Mapper;

class User
{
        private $mapping = array(
                'id' => 'id',
                'firstName' => 'first_name',
                'lastName' => 'last_name',
                'gender' => 'gender',
                'namePrefix' => 'name_prefix'
        );

        public function extract($user)
        {
                $data = array();

                foreach($this->mapping as $keyObject => $keyColumn) {

                if ($keyColumn != 'id') {
                   $data[$keyColumn] = call_user_func(
                       array($user, 'get'. ucfirst($keyObject))
                   );
                }
           }

           return $data;
        }

        public function populate($data, $user)
        {
                $mappingsFlipped = array_flip($this->mapping);

                foreach($data as $key => $value) {
                        if(isset($mappingsFlipped[$key])) {
                                call_user_func_array(
                                        array($user, 'set'. ucfirst(
                                        $mappingsFlipped[$key])),
                                        array($value)
                                );
                        }
```

```
41                  }
42
43                  return $user;
44          }
45  }
```

This is how we extract the data from the object. The `EntityManager`, extended by a `saveUser()` method, now can insert a new record into the database:

```
 1  <?php
 2
 3  include_once('../repository/User.php');
 4  include_once('../mapper/User.php');
 5
 6  use Repository\User as  UserRepository;
 7  use Mapper\User as UserMapper;
 8
 9  class EntityManager
10  {
11          private $host;
12          private $db;
13          private $user;
14          private $pwd;
15          private $connection;
16          private $userRepository;
17
18          public function __construct($host, $db, $user, $pwd)
19          {
20                  $this->host = $host;
21                  $this->user = $user;
22                  $this->pwd = $pwd;
23
24                  $this->connection =
25                          new \PDO("mysql:host=$host;dbname=$db", $user,
                            $pwd);
26
27                  $this->userRepository = null;
28          }
29
30          public function query($stmt)
31          {
32                  return $this->connection->query($stmt);
33          }
34
35          public function saveUser($user)
36          {
37                  $userMapper = new UserMapper();
38                  $data = $userMapper->extract($user);
```

```
39                  $columnsString = implode(", ", array_keys($data));
40
41                  $valuesString = implode(
42                          "',
43                          '",
44                          array_map("mysql_real_escape_string", $data)
45                  );
46
47                  return $this->query(
48                          "INSERT INTO users ($columnsString)
                            VALUES('$valuesString')"
49                  );
50          }
51
52          public function getUserRepository()
53          {
54                  if (!is_null($this->userRepository)) {
55                  return $this->userRepository;
56                  } else {
57                          $this->userRepository = new UserRepository($this);
58                          return $this->userRepository;
59                  }
60          }
61  }
```

Adding a new record now works like this:

```
1   <?php
2   include_once('../EntityManager.php');
3   $em = new EntityManager('localhost', 'app', 'root', '');
4   $user = $em->getUserRepository()->findOneById(1);
5   echo $user->assembleDisplayName() . '<br />';
6
7   $newUser = new Entity\User();
8   $newUser->setFirstName('Ute');
9   $newUser->setLastName('Mustermann');
10  $newUser->setGender(1);
11  $em->saveUser($newUser);
12
13  echo $newUser->assembleDisplayName();
```

So far, so good! But what if we want to update an existing record? How do we identify whether we are dealing with a new record or one that already exists? In our case, we might simply check to see whether the object already has a value for the given `ID` field. `ID` is an "auto_increment" field, so MySQL will populate it automatically for any new record. For sure, this is not the most elegant solution one might come up with. We might use a so-called *identity map* that brings more advantages, such as re-reading an already-loaded

database without querying the database again. An identity map is nothing more than an associative array holding references to already loaded entities based on IDs. A good place for an identity map is the entity manager and its saveUser() method:

```
<?php

include_once('../repository/User.php');
include_once('../mapper/User.php');

use Repository\User as UserRepository;
use Mapper\User as UserMapper;

class EntityManager
{
        private $host;
        private $db;
        private $user;
        private $pwd;
        private $connection;
        private $userRepository;
        private $identityMap;

        public function construct($host, $db, $user, $pwd)
        {
                $this->host = $host;
                $this->user = $user;
                $this->pwd = $pwd;

                $this->connection =
                        new \PDO("mysql:host=$host;dbname=$db", $user,
                        $pwd);

                $this->userRepository = null;
                $this->identityMap = array('users' => array());
        }

        public function query($stmt)
        {
                return $this->connection->query($stmt);
        }

        public function saveUser($user)
        {
                $userMapper = new UserMapper();
                $data = $userMapper->extract($user);

                $userId = call_user_func(
```

```
43                          array($user, 'get'. ucfirst($userMapper->
                            getIdColumn()))
44                  );
45
46                  if (array_key_exists($userId, $this->identityMap['users'])) {
47                  $setString = '';
48
49                          foreach ($data as  $key => $value) {
50                                  $setString .= $key."='$value',";
51                          }
52
53                          return $this->query(
54                                  "UPDATE users SET " . substr($setString,
                                    0, -1) .
55                                  " WHERE " . $userMapper->getIdColumn() .
                                    "=" . $userId
56                          );
57
58                  } else {
59                          $columnsString = implode(", ", array_keys($data));
60                          $valuesString = implode(
61                                  "',
62                                  '",
63                                  array_map("mysql_real_escape_string",
                                    $data)
64                          );
65
66                          return $this->query(
67                                  "INSERT INTO users ($columnsString)
                                    VALUES('$valuesString')"
68                          );
69                  }
70          }
71
72          public function getUserRepository()
73          {
74                  if (!is_null($this->userRepository)) {
75                          return $this->userRepository;
76                  } else {
77                          $this->userRepository = new UserRepository($this);
78                          return $this->userRepository;
79                  }
80          }
81
82          public function registerUserEntity($id, $user)
83          {
84                  $this->identityMap['users'][$id] = $user;
```

```
85                  return $user;
86          }
87  }
```

As you can see, we added a method, `getIdColumn()`, which returns "`id`." Now the following code works nicely:

```
1   <?php
2   include_once('../EntityManager.php');
3   $em = new EntityManager('localhost', 'app', 'root', '');
4   $user = $em->getUserRepository()->findOneById(1);
5   echo $user->assembleDisplayName() . '<br />';
6
7   $user->setFirstname('Moritz');
8   $em->saveUser($user);
```

The entity is updated in the database and no additional record is added.

Associations

Now we want to list all posts from a specific `User`. To do this, we would like to simply iterate over the `Posts` collection of a given `User` and print each `Post`'s title:

```
 1   <?php
 2   include_once('../EntityManager.php');
 3   $em = new EntityManager('localhost', 'app', 'root', '');
 4   $user = $em->getUserRepository()->findOneById(1);
 5   ?>
 6   <h1><?php echo $user->assembleDisplayName(); ?></h1>
 7   <ul>
 8   <?php foreach($user->getPosts() as $post) { ?>
 9   <li><?php  echo $post->getTitle(); ?></li>
10   <?php } ?>
11   </ul>
```

What do we have to do to make this happen?
First, we create a data structure for posts:

```
1   CREATE TABLE posts(
2           id int(10) NOT NULL auto_increment,
3           user_id int(10) NOT NULL,
4           title varchar(255) NOT NULL,
5           content text NOT NULL,
6           PRIMARY KEY (id)
7   );
```

We add a test User and some test Posts, so that we can actually test the implementation. Next, we need to extend a whole bunch of classes. The User entity gets a getPosts() method, which loads the User's Posts on first invocation via the corresponding repository:

```
1   <?php
2   namespace Entity;
3
4   class User
5   {
6           // [..]
7
8      private $postRepository;
9
10     public function getPosts()
11         {
12             if (is_null($this->posts)) {
13                 $this->posts = $this->postRepository->findByUser($this);
14             }
15
16             return $this->posts;
17         }
18
19         // [..]
20  }
```

This will work only if the User entity has access to the Post repository. To make it available, the User repository method findOneById() needs to be extended:

```
1   <?php
2   namespace Repository;
3
4   include_once('../entity/User.php');
5   include_once('../mapper/User.php');
6
7   use Mapper\User as UserMapper;
8   use Entity\User as UserEntity;
9
10  class User
11  {
12          private $em;
13          private $mapper;
14
15          public function __construct($em)
16          {
17                  $this->mapper = new UserMapper;
18                  $this->em = $em;
19          }
20
```

```
21          public function findOneById($id)
22          {
23                  $userData = $this->em
24                          ->query('SELECT * FROM users WHERE id = ' . $id)
25                          ->fetchAll();
26
27                  $newUser = new UserEntity();
28                  $newUser->setPostRepository(
                    $this->em->getPostRepository());
29
30                  return $this->em->registerUserEntity(
31                          $id,
32                          $this->mapper->populate($userData, $newUser)
33                  );
34          }
35  }
```

The entity manager needs to be extended by the method `getPostRepository()` as well:

```
1   <?php
2
3   // [..]
4
5   class EntityManager
6   {
7           // [..]
8
9           private $postRepository;
10
11          public function getPostRepository()
12          {
13                  if (!is_null($this->postRepository)) {
14                          return $this->postRepository;
15                  } else {
16                          $this->postRepository = new
                            PostRepository($this);
17                          return $this->postRepository;
18                  }
19          }
20  }
```

Now, the `Post` entity and the `Post` mapper must be implemented:

```
1   <?php
2   namespace Entity;
3
4   class Post
```

```
{
        private $id;
        private $title;
        private $content;

        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;
        }
}
```

And here is the mapper:

```
<?php
namespace Mapper;

class Post
{
        private $mapping = array(
                'id' => 'id',
                'title' => 'title',
                'content' => 'content',
        );
```

```
11
12          public function getIdColumn()
13          {
14                  return 'id';
15          }
16
17          public function extract($user)
18          {
19                  $data = array();
20
21                  foreach ($this->mapping as $keyObject => $keyColumn) {
22                          if ($keyColumn != $this->getIdColumn()) {
23                                  $data[$keyColumn] = call_user_func(
24                                          array($user, 'get'.
                                            ucfirst($keyObject))
25                                  );
26                          }
27                  }
28
29                  return $data;
30          }
31
32          public function populate($data, $user)
33          {
34                  $mappingsFlipped = array_flip($this->mapping);
35
36                  foreach ($data as $key => $value) {
37                          if (isset($mappingsFlipped[$key])) {
38                                  call_user_func_array(
39                                          array($user, 'set'. ucfirst(
                                            $mappingsFlipped[$key])),
40                                          array($value)
41                                  );
42                          }
43                  }
44
45                  return $user;
46          }
47  }
```

Last but not least, the Post repository:

```
1   <?php
2   namespace Repository;
3
4   include_once('../entity/Post.php');
5   include_once('../mapper/Post.php');
6
```

```
use Mapper\Post as PostMapper;
use Entity\Post as PostEntity;

class Post
{
        private $em;
        private $mapper;

        public function __construct($em)
        {
                $this->mapper = new PostMapper;
                $this->em = $em;
        }

        public function findByUser($user)
        {
                $postsData = $this->em
                        ->query('SELECT * FROM posts WHERE user_id = '
                        . $user->getId())
                        ->fetchAll();

                $posts = array();

                foreach($postsData as $postData) {
                        $newPost = new PostEntity();
                        $posts[] = $this->mapper->populate($postData,
                        $newPost);
                }

                return $posts;
        }
}
```

That's it! Up to this point, the application's files and folder structure looks like this:

```
EntityManager.php
entity/
        Post.php
        User.php
mapper/
        Post.php
        User.php
repository/
        Post.php
        User.php
public/
        index.php
```

Next Steps

Great! We built our own little ORM tool. However, already, our code doesn't look that good anymore. The fact that technical code is mixed up with domain-specific code is an issue. Our solutions to the problems faced are valid only to our concrete use case. Also, it smells like "copy & paste" in here! In fact, some refactoring already needs to be done.

And what about composite primary keys? Adding and deleting associations? Many-to-many associations? Inheritance, performance, caching and entities, mappers and repositories for the tons of yet-missing core elements of the application? Also, what happens if the data structures change? This would mean refactoring of multiple classes! Looks like it is time for Doctrine 2 to enter the stage.

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):

```
1   {
2         "require": {
3                 "doctrine/orm": "2.3.1"
4          }
5   }
```

In the root directory of the application, execute the command

```
1   $ 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
// [..]
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
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;

```

```
25          /** @Column(type="string", name="name_prefix", nullable=true)  */
26          private $namePrefix;
27
28          const GENDER_MALE = 0;
29          const GENDER_FEMALE = 1;
30
31          const GENDER_MALE_DISPLAY_VALUE = "Mr.";
32          const GENDER_FEMALE_DISPLAY_VALUE = "Ms.";
33
34          public function assembleDisplayName()
35          {
36                  $displayName = '';
37
38                  if ($this->gender == self::GENDER_MALE) {
39                          $displayName .= self::GENDER_MALE_DISPLAY_VALUE;
40                  } elseif ($this->gender == self::GENDER_FEMALE) {
41                          $displayName .= self::GENDER_FEMALE_DISPLAY_
                            VALUE;
42                  }
43
44                  if ($this->namePrefix) {
45                          $displayName .= ' ' . $this->namePrefix;
46                  }
47
48                  $displayName .= ' ' . $this->firstName . ' ' . $this->lastName;
49
50                  return $displayName;
51          }
52
53          public function setFirstName($firstName)
54          {
55                  $this->firstName = $firstName;
56          }
57
58          public function getFirstName()
59          {
60                  return $this->firstName;
61          }
62
63          public function setGender($gender)
64          {
65                  $this->gender = $gender;
66          }
67
68          public function getGender()
69          {
70                  return $this->gender;
71          }
```

```
72
73          public function setId($id)
74          {
75                  $this->id = $id;
76          }
77
78          public function getId()
79          {
80                  return $this->id;
81          }
82
83          public function setLastName($lastName)
84          {
85                  $this->lastName = $lastName;
86          }
87
88          public function getLastName()
89          {
90                  return $this->lastName;
91          }
92
93          public function setNamePrefix($namePrefix)
94          {
95                  $this->namePrefix = $namePrefix;
96          }
97
98          public function getNamePrefix()
99          {
100                 return $this->namePrefix;
101         }
102  }
```

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

```
1   <?php
2   include '../entity/User.php';
3   include '../vendor/autoload.php';
4
5   use Doctrine\ORM\Tools\Setup;
6   use Doctrine\ORM\EntityManager;
7
8   $paths = array(__DIR__ . "/../entity/");
9   $isDevMode = true;
10
11  $dbParams = array(
12          'driver' => 'pdo_mysql',
13          'user' => 'root',
```

```
14          'password' => '',
15          'dbname' => 'app',
16  );
17
18  $config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
19  $em = EntityManager::create($dbParams, $config);
20  $user = $em->getRepository('Entity\User')->findOneById(1);
21  echo $user->assembleDisplayName() . '<br />';
22  $user->setFirstname('Moritz');
23  $em->persist($user);
24  $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:

```
1   <?php
2   namespace Entity;
3
4   /**
5    * @Entity
6    * @Table(name="users")
7    */
8   class User
9   {
10          // [..]
11
12          /**
13          * @OneToMany(targetEntity="Entity\Post", mappedBy="user")
14          */
15          private $posts;
16
17          // [..]
18
19          public function __construct()
20          {
21                  $this->posts = new \Doctrine\Common\Collections\
                    ArrayCollection();
22          }
23
24          // [..]
25  }
```

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;
        }

```

```
48          public function setId($id)
49          {
50                  $this->id = $id;
51          }
52
53          public function getId()
54          {
55                  return $this->id;
56          }
57
58          public function setTitle($title)
59          {
60                  $this->title = $title;
61          }
62
63          public function getTitle()
64          {
65                  return $this->title;
66          }
67  }
```

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

```
1   <?php
2   // [..]
3   $user = $em->getRepository('Entity\User')->findOneById(1);
4   ?>
5   <h1><?echo $user->assembleDisplayName(); ?></h1>
6   <ul>
7           <?php foreach($user->getPosts() as $post) {?>
8                   <li><?php echo $post->getTitle(); ?></li>
9           <?php } ?>
10  </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:

```
1   entity/
2           Post.php
3           User.php
4   public/
5           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!

CHAPTER 5

Defining Entities

We have already learned how easy it is to map member variables to fields in a database table. Let's take a look at exactly how this works.

Mapping Formats

In general, mappings can be defined using annotations within the entities themselves, or with external XML or YAML files. In terms of performance, there is no difference between the different ways of implementing the mapping when using a cache. Therefore, using annotations is often preferred, simply because they make coding easier, as all the persistence metadata is located in the same file. In the following examples, annotations are used exclusively.

Annotations are meta information used to tell Doctrine how to persist data. Syntactically, Doctrine annotations follow the PHPDoc standard which is also used to auto-generate code documentation using compatible tools. PHPDoc uses individual DocBlocks to provide meta information on certain code elements. In several IDEs, annotations are also used to provide improved code completion, type hinting and to support debugging.

Mapping Objects to Tables

First of all, a class that acts as a template for persistent objects needs to have the `@Entity` annotation:

```
1   <?php
2   /**
3   * @Entity
4   */
5   class User
6   {
7           // [..]
8   }
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_5

A DocBlock is an extended PHP comment that begins with "/**" and has an "*" at the beginning of every line. Simply adding `@Entity` without additional configuration means that Doctrine 2 looks for a database table named `user`, in this case. If the table has another name, the table name can be configured like this:

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

An optional attribute, "`indexes`," of the `@Table` annotation can be used to create indexes automatically using Doctrine's schema tool, which we will discuss later in the book. For now, one should remember that this attribute has no meaning during runtime. Some persistence metadata is important only in conjunction with offline tools such as the schema tool, while other metadata is evaluated during runtime. An index definition to the `users` table goes like this:

```
 1   <?php
 2   /**
 3   * @Entity
 4   * @Table(name="users",
 5   *        indexes={@Index(name="search_idx", columns={"name", "email"})}
 6   * )
 7   */
 8   class User
 9   {
10          // [..]
11   }
```

The `indexes` attribute of annotation `@Table` itself has an annotation with attributes as its value. The annotation `@Index` needs the `name` attribute (name of the index) and the list of `columns` to be considered. Multiple values may be given using curly braces.

The same applies to another optional attribute called `uniqueConstraints` which defines special unique value indexes:

```
.1   <?php
 2   /**
 3   * @Entity
 4   * @Table(name="users",
 5   *         uniqueConstraints={@UniqueConstraint(name="user_unique",
 6   *                 columns={"username"})},
```

```
 7   *        indexes={@Index(name="user_idx", columns={"email"})}
 8   * )
 9   */
10   class User
11   {
12           // [..]
13   }
```

Mapping Scalar Member Variables to Fields

Mapping scalar-valued member variables (numbers or strings, for example) to a database table is super simple:

```
1   <?php
2
3   / [..]
4
5   /** @Column(type="string") */
6   private $lastName;
```

First, it's important that all member variables mapped to database fields are declared as `private` or `protected`. Doctrine 2 often applies a technique called *lazy loading*, which means that data is loaded just in the moment it is needed, but not earlier. To make this happen, Doctrine 2 implements so-called proxy objects which rely on the entities' getter methods. Therefore, one will want to make sure they are set-up.

The main annotation here is `@Column`. Its attribute type is the only mandatory attribute and is important to make persistence work correctly. If a wrong type is defined, you may lose information while saving to or reading from the database. In addition, there are many other optional attributes, such as `name`, which allows you to configure a table column name. It defaults to the name of the member variable. The `length` attribute is needed only for strings, where it defines the maximum length of a string value. It defaults to 255 if not defined differently. If one encounters truncated values in the database, a wrong length configuration often is the root cause.

The attributes `precision` and `scale` are valid only for values of type `decimal`, while `unique` ensures uniqueness, and `nullable` tells Doctrine 2 whether a `NULL` value is allowed (`true`) or not (`false`):

```
1   <?php
2
3   // [..]
4
5   /**
6   * @Column(type="string",
7   *         name="last_name", length=32, unique=true, nullable=false)
8   */
9   protected $lastName;
```

Data Types

Doctrine 2 ships with a set of data types that map PHP data types to SQL data types. Doctrine 2 data types are compatible with most common database systems:

`string`: Type that maps an SQL `VARCHAR` to a PHP `string`

`integer`: Type that maps an SQL `INT` to a PHP `integer`

`smallint`: Type that maps a database `SMALLINT` to a PHP `integer`

`bigint`: Type that maps a database `BIGINT` to a PHP `string`

`boolean`: Type that maps an SQL `boolean` to a PHP `boolean`

`decimal`: Type that maps an SQL `DECIMAL` to a PHP `double`

`date`: Type that maps an SQL `DATETIME` to a PHP `DateTime` object

`time`: Type that maps an SQL `TIME` to a PHP `DateTime` object

`datetime`: Type that maps an SQL `DATETIME/TIMESTAMP` to a PHP `DateTime` object

`text`: Type that maps an SQL `CLOB` to a PHP `string`

`object`: Type that maps an SQL `CLOB` to a PHP `object` using `serialize()` and `unserialize()`

`array`: Type that maps an SQL `CLOB` to a PHP `object` using `serialize()` and `unserialize()`

`float`: Type that maps an SQL Float (Double Precision) to a PHP `double`. IMPORTANT: Type `float` works only with locale settings that use decimal points as separators.

Custom data types If needed, you can define custom data types, mappings between PHP data types and SQL data types. The official documentation covers this topic in depth.[1]

[1] http://docs.doctrine-project.org/en/latest/reference/basic-mapping.html#custom-mapping-types

Entity Identifier

An entity class must define a unique identifier. The `@Id` annotation is used for this:

```
1   <?php
2
3   // [..]
4
5   /**
6   * @Id @Column(type="integer")
7   */
8   private $id;
```

If the `@GeneratedValue` annotation is used as well, the unique identifier doesn't need to be given by the application developer but is assigned automatically by the database system. The way this value is generated depends on the strategy defined. By default, Doctrine 2 identifies the best approach on its own. When using MySQL, for instance, Doctrine 2 uses the `auto_increment` function. There are additional strategies that can be used; however, they usually don't work across platforms and therefore should be avoided.

By the way, the `@Id` annotation can be used with multiple member variables to define composite keys. Clearly, the `@GeneratedValue` annotation then cannot be used anymore. In this case, the application developer needs to take care in assigning unique keys.

Inheritance

Doctrine 2 supports persistence of hereditary structures with three different strategies:

- Single Table Inheritance
- Class Table Inheritance
- Mapped Superclass

Single Table Inheritance

The *single table inheritance* strategy uses a single database table with a so-called "discriminator column" to differentiate between the different types. All classes of a hierarchy go into the same table. Let's say our demo application differentiates between different types of posts. The supported types are:

`Post`: A simple, text based post

`ImagePost`: A post with text and image

`VideoPost`: A post with text and video

Both ImagePost and VideoPost extend Post and add an image or video URL. To set up this hierarchy, the topmost Post class holds the configuration needed as well as all member variables and associations that are shared between the three types:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
* @InheritanceType("SINGLE_TABLE")
* @DiscriminatorColumn(name="discr", type="string")
* @DiscriminatorMap({"text"="Post","video"="VideoPost","image"=
"ImagePost"})
*/
class Post
{
        // member variables and associations shared between all types
}
```

Since the Post itself can be an entity, it has the typical entity-related annotations. In addition, via the @InheritanceType annotation, it configures single table inheritance. The @DiscriminatorColumn annotation is used to label the column which indicates the type persisted. Lastly, @DiscriminatorMap defines the values that can be present in the discriminator column. In our case, Doctrine 2 writes "text" into the discriminator column if the entity is a Post. If the entity is a VideoPost, Doctrine 2 writes "video," and if the entity is an ImagePost, the value used is "image."

The ImagePost itself is straightforward:

```
<?php
namespace Entity;

/**
* @Entity
*/
class ImagePost extends Post
{
        /** @Column(type="string") */
        protected $imageUrl;

        /**
        * @param mixed $imageUrl
        */
        public function setImageUrl($imageUrl)
        {
                $this->imageUrl = $imageUrl;
        }

```

```
20          /**
21          * @return mixed
22          */
23          public function getImageUrl()
24          {
25                  return $this->imageUrl;
26          }
27  }
```

The same is true for the VideoPost:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   */
7   class VideoPost extends Post
8   {
9           /** @Column(type="string") */
10          protected $videoUrl;
11
12          /**
13          * @param mixed $videoUrl
14          */
15          public function setVideoUrl($videoUrl)
16          {
17                  $this->videoUrl = $videoUrl;
18          }
19
20          /**
21          * @return mixed
22          */
23          public function getVideoUrl()
24          {
25                  return $this->videoUrl;
26          }
27  }
```

The data structure used by Doctrine 2 looks like this:

Col	Type	Length	NULL?	KEY	Extras
Id	INT	11		PRI	auto_ increment
user_id	INT	11	YES	MUL	
category_id	INT	11	YES	MUL	
title	VARCHAR	255			
content	LONGTEXT				
discr	VARCHAR	255			
videoUrl	VARCHAR	255	YES		
imageUrl	VARCHAR	255	YES		

Now all three types of entities can easily be persisted. The downside of this approach is that, depending on how complex the hierarchy is and how different the types involved are, the table grows and grows, with many fields never used by certain types. In our case, the `videoUrl` won't ever be used by entities of type `ImagePost`. The `imageUrl` won't be used by `VideoPost` entities, and for `Post` entities, both fields will remain blank.

The upside is that this strategy—due to its very nature—allows for very efficient queueing across all types in the hierarchy without any joins.

Class Table Inheritance

While single table inheritance uses only one table for all types, with *class table inheritance* each class in the hierarchy is mapped to several tables. While this gives the best flexibility of the three strategies, this also means that reconstructing entities will require joins in the database. Since joins are usually expensive operations, the performance with class table inheritance is not as good as with single table inheritance.

The main difference in configuration is in `Post`:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity(repositoryClass="Repository\Post")
6  * @Table(name="posts")
7  * @InheritanceType("JOINED")
8  * @DiscriminatorColumn(name="discr", type="string")
9  * @DiscriminatorMap({"text"="Post","video"="VideoPost","image"=
   "ImagePost"})
10 */
```

```
11  class Post
12  {
13          // member variables and associations shared between all types
14  }
```

Instead of inheritance type "SINGLE_TABLE," now "JOINED" is used. This tells Doctrine 2 to apply the class table inheritance strategy. Since tables will be used for each type, configuring the table names may be useful for subclasses:

```
 1  <?php
 2  namespace Entity;
 3
 4  /**
 5  * @Entity
 6  * @Table(name="posts_image")
 7  */
 8  class ImagePost extends Post
 9  {
10          /** @Column(type="string") */
11          protected $imageUrl;
12
13          /**
14          * @param mixed $imageUrl
15          */
16          public function setImageUrl($imageUrl)
17          {
18                  $this->imageUrl = $imageUrl;
19          }
20
21          /**
22          * @return mixed
23          */
24          public function getImageUrl()
25          {
26                  return $this->imageUrl;
27          }
28  }
```

The table name `posts_image` is handy for `ImagesPost` entities. We add the same style of configuration to `VideoPost` entities as well:

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

```
 7   */
 8   class VideoPost extends Post
 9   {
10           /** @Column(type="string") */
11           protected $videoUrl;
12
13           /**
14           * @param mixed $videoUrl
15           */
16           public function setVideoUrl($videoUrl)
17           {
18                   $this->videoUrl = $videoUrl;
19           }
20
21           /**
22           * @return mixed
23           */
24           public function getVideoUrl()
25           {
26                   return $this->videoUrl;
27           }
28   }
```

When letting Doctrine 2 create the database schema, we will end up with three tables for this hierarchy, while the `posts` table holds the main information:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	auto_ increment
user_id	INT	11	YES	MUL	
category_id	INT	11	YES	MUL	
title	VARCHAR	255			
content	LONGTEXT				
discr	VARCHAR	255			

The `posts_video` table holds only the specifics of this type:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
videoUrl	VARCHAR	255	YES		

The same is true for `posts_image`:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
imageUrl	VARCHAR	255	YES		

By evaluating the foreign key reference, entities of type `ImagePost` and `VideoPost` can be reconstructed by table joins.

Mapped Superclass

Lastly, a *mapped superclass* strategy can be used. The main difference from the other two strategies already discussed is that the superclass itself cannot be an entity. Taking the example above, this means that class `Post` cannot itself be an entity. It simply provides member variables and associations to be inherited by its subclasses.

To configure the mapped superclass strategy means adding the `@MappedSuperclass` annotation to the `Post` class:

```
1   <?php
2   namespace Entity;
3
4   /** @MappedSuperclass */
5   class Post
6   {
7                   // member variables and associations shared between all types
8   }
```

The two subclasses, `ImagePost` and `VideoPost`, do not need to be changed. However, with the mapped superclass strategy applied, we cannot persist objects of type `Post` anymore, which mainly consisted of text. We could fix this by adding another new type called `TextPost`:

```
 1   <?php
 2   namespace Entity;
 3
 4   /**
 5   * @Entity
 6   * @Table(name="posts_text")
 7   */
 8   class TextPost extends Post
 9   {
10           /**
11           * @Id @Column(type="integer")
12           * @GeneratedValue
13           */
```

```
14          protected $id;
15
16  }
```

Now we have three concrete entity types and one superclass that is not an entity. This results in three tables in the database: `posts_text`, `posts_images` and `posts_video`.

As shown above, it's important that each entity specifies its `ID` column, while the mapped superclass does not. It only imparts member variables and associations, but not the ID column definition.

When inheriting attributes or associations as shown above, one may want to overwrite certain definitions.

Let's assume we don't want to talk about `content` in a `VideoPost`, but call it `closed_caption`. This requires the following configuration:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="posts_video")
7   * @AttributeOverrides({
8   *     @AttributeOverride(name="content",
9   *         column=@Column(
10  *             name = "closed_caption",
11  *             type = "text"
12  *         )
13  *     )
14  * })
15  */
16  class VideoPost extends Post
17  {
18          // [..]
19  }
```

While Doctrine 2 requires member variables to be either `private` or `protected`, it's important to remember that member variables need to be `protected` here; otherwise they won't be available in the subclass. When overwriting the property `content` with `closed_caption` as shown above, this leads to a `closed_caption` column in the database for this type of entity.

Also, entities of type `VideoPost` are organized in categories called `channels`. We can overwrite the association as shown below:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
```

```
 * @Table(name="posts_video")
 * @AssociationOverrides({
 *     @AssociationOverride(name="category",
 *          joinColumns=@JoinColumn(
 *              name="channel_id",  referencedColumnName="id"
 *        )
 *    )
 * })
 */
class VideoPost extends Post
{
          // [..]
}
```

Summary

In this chapter, we learned all about entities, the most important aspect of ORM and Doctrine. Designing entities properly can be fiddly at times, but Doctrine makes it as simple as possible to lay out the foundation for persisting PHP objects. Since entities alone already are a great thing to have, it get's even better, once we start interlinking them by using associations, which we will cover next.

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.

CHAPTER 7

Managing Entities

Creating a New Entity

Once the domain model has been constructed, it is time to use it. A new entity can be created based simply on a new object of an entity class:

```
1   <?php
2   $newPost = new \Entity\Post();
3   $newPost->setTitle('A new post!');
4   $newPost->setContent('This is the body of the new post.');
5   $em->persist($newPost);
6   $em->flush();
```

The code shown above anticipates that `$em` references a ready-to-use entity manager. First, a new `Post` is created, then data is assigned, and the object is passed to the entity manager for persistence.

One must remember that the `persist()` method call does not yet cause an SQL `INSERT` statement to be issued. The entity is only scheduled for persistence with the next flushing. As long as no flushing has taken place, the entity is in a state called `MANAGED`, meaning that the entity manager recognizes the new entity.

Only when the `flush()` method is invoked on the entity manager is a new record written to the database. Otherwise, the entity will be lost after the script has finished.

Loading an Existing Entity

There are two main ways to load an existing entity: either by querying and retrieving it from its corresponding repository or by accessing it through an association given by another, already loaded entity.

Using a Repository

We already learned that a repository is a container for all entities of a specific type. A repository provides finder methods to search for entities based on a query. While several finder methods are available out-of-the-box, custom finder methods can also be added later on. A custom finder method can be imagined as a "quick access" to a typical query. With the help of finder methods, you can look up an entity, for example by its ID, like this:

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_7

```
<?php
$post = $em->getRepository('Entity\Post')->findOneById($id);
```

First we request the repository from the entity manager and then we execute a finder method on it. By default, four finder methods are available. To find a single entity based on its ID, use the `find()` method:

```
<?php
public function find($id, $lockMode = LockMode::NONE, $lockVersion =
null);
```

To find a single entity based on criteria, use the `findOneBy()` method:

```
<?php
public function findOneBy(array $criteria, array $orderBy = null);
```

To find all entities of a specific type, use the `findAll()` method:

```
<?php
public function findAll();
```

To find multiple entities based on criteria, use the `findBy()` method:

```
<?php
public function findBy(
        array $criteria,
        array $orderBy = null,
        $limit = null,
        $offset = null
);
```

You can define the order, limit, and offset values when using `findBy()`.

Also helpful are the so-called *magic finders*. They allow you to include search criteria directly in a finder method's name. The next two statements produce the same result:

```
<?php
$tag = $em->getRepository('Entity\Tag')->findOneBy(array('label'=>$label));
$tag = $em->getRepository('Entity\Tag')->findOneByLabel($label);
```

Adding a custom repository with individual finder methods is a two-step process. First, a new repository class needs to be set up:

```
<?php
namespace Repository;

use Doctrine\ORM\EntityRepository;

```

```
 6   class Post extends EntityRepository
 7   {
 8           public function findAllPostsWithTag($tag)
 9           {
10                   // DQL statement goes here
11           }
12   }
```

Next, Doctrine 2 needs to know about the new repository. This is done through the corresponding entity:

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

That's all! The `findAllPostsWithTag()` finder method can now be easily invoked:

```
1   <?php
2   $posts = $em->getRepository('Entity\Post')->findAllPostsWithTag($tag);
```

We will learn more about DQL, the *Doctrine Query Language*, later in the book. It is used to phrase a query.

Using an Association

Let's assume we already have a loaded `User` entity available. Instead of loading the `User's` contact data by using its repository, we can also reach this entity from the given `User` entity:

```
 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          */
15          private $contactData;
16
17          // [..]
18
19          public function getContactData()
20          {
21                  return $this->contactData;
22          }
23   }
```

In this case, when calling `getContactData()`, the referenced `ContactData` entity is loaded on demand by Doctrine's proxy mechanism.

We could even make sure that the referenced `ContactData` entity is already loaded when loading the `User` itself, and save an additional database query:

```
1   <?php
2   // [..]
3   /**
4   * @OneToOne(targetEntity="Entity\ContactData", fetch="EAGER")
5   */
6   private $contactData;
```

By using `fetch="EAGER,"` we tell Doctrine 2 to always load the referenced `ContactData` entity when the `User` itself is loaded.

Loading eagerly sometimes has its advantages, especially when it's likely to access a referenced entity later in the process. If the `fetch` attribute is omitted, Doctrine 2 fetches `LAZY` by default. This means it loads the entity on first access. The third option is `EXTRA_LAZY`, which is helpful for huge datasets. Even if one decides to lazy load references, the referenced entities are still all loaded fully into RAM. Depending on the amount and size of the entities referenced, this could be a serious performance issue. When `EXTRA_LAZY` is used, several methods can be executed on the collection of referenced entities without fully loading them into the RAM right away. This is true for:

- `contains()`
- `count()`
- `offsetSet()`
- `add()`
- `count()`
- `slice()`

In this way, a pagination feature, for example, can be built without performance issues.

Changing an Existing Entity

Modifying an existing, already loaded entity is easy. All changes made to such an entity are auto-detected by Doctrine 2 when flushing the entity manager:

```
1   <?php
2   // [..]
3   $post = $em->getRepository('Entity\Post')->find(1);
4   $post->setTitle("New title");
5   $em->flush();
```

There is no need for explicitly telling Doctrine 2 again about the fact that this entity has been changed. The method `persist($entity)` does not need to be called on the entity manager again.

Removing an Entity

Removing an existing entity can easily be done through the entity manager, if a handle to a loaded entity is available:

```
1   <?php
2   $em->remove($post);
3   $em->flush();
```

The SQL statement needed to physically delete the record in the database is not issued as long as `flush()` has not yet been invoked.

Sorting an Association

When accessing entities via an association, the order of the entities retrieved is not defined. As a part of the entity mapping definitions, one can define the order of entities using the `@OrderBy` annotation. If we get back to our demo application, we could define the order of a `User's Posts` in such a way that the most recent `Post` is shown first in its list of `Posts`, simply by modifying the mapping configuration in the `User` entity:

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

```
11
12          /**
13          * @OneToMany(targetEntity="Entity\Post", mappedBy="user")
14          * @OrderBy({"id" = "DESC"})
15          */
16          private $posts;
17
18          // [..]
19      }
```

Alternatively, you can sort a collection usng PHP after retrieving the entities from the database.

Removing an Association

Removing an association is as straightforward as removing an entity:

```
1   <?php
2   $newPost = new \Entity\Post();
3   $newPost->setTitle('A new post!');
4   $newPost->setContent('This is the body of the new post.');
5   $user = $em->getRepository('Entity\User')->findOneById(1);
6   $newPost->setUser($user);
7   $em->flush();
8
9   $newPost->setUser(null);
10  $em->flush();
```

In the example above, a new `Post` entity is created. Then, an existing `User` entity is loaded and associated with the new post. After flushing, the reference has been persisted to the database. Next, we remove the association again by setting `User` to `null`. After the next flushing, the reference is gone in the database.

We need to keep in mind here that we have established a bidirectional relationship between the `User` and its `Posts`, and the `Post` entity is the owning side of the association. If, for example, we would take the `User` entity and remove the `Post` from its collection `$posts`, nothing would happen on flushing:

```
1   <?php
2   // [..]
3   $user->getPosts()->removeElement($newPost);
4   $em->flush();
```

The `removeElement()` method, which is used to remove an element from a given Doctrine 2 collection based on an entity loaded, is without the desired effect. However, there *is* an effect. While the change won't be persisted, the element has been removed from the collection in RAM. One won't find the element anymore when looking it up in the collection.

Lifecycle Events

When working with entities, several events are triggered by Doctrine 2:

`preRemove`: Occurs for a given entity before the respective `EntityManager remove` operation for that entity is executed.

`postRemove`: Occurs for an entity after the entity has been deleted. It will be invoked after the database `delete` operations.

`prePersist`: Occurs for a given entity before the respective `EntityManager persist` operation for that entity is executed.

`postPersist`: Occurs for an entity after the entity has been made persistent. It will be invoked after the database `insert` operations.

`preUpdate`: Occurs before the database `update` operations to entity data.

`postUpdate`: Occurs after the database `update` operations to entity data.

`postLoad`: Occurs for an entity after the entity has been loaded from the database.

With `loadClassMetadata, onFlush,` and `onClear`, additional events are triggered that are persistence-related, but are not specific to a single entity.

With these events available, you can hook into persistence processing, with a so-called *lifecycle callback*, which is implemented as a method of an entity class.

Let's assume we want to add login data for each `User` to our demo application. While the username can be picked by the user, the password is auto-generated on signup. This can be achieved by adding a lifecycle callback to the `User` class:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @HasLifecycleCallbacks
7   * @Table(name="users")
8   */
9   class User
10  {
11          // [..]
12
13          const GENERATED_PASSWORD_LENGTH = 6;
14
15          // [..]
16
```

```
17          /** @PrePersist */
18          public function generatePassword()
19          {
20                  for($i = 1; $i <= self::GENERATED_PASSWORD_LENGTH;
                    $i++) {
21                          $this->password .= chr(rand(65, 90)); // 65 ->
                            A, 90 -> Z
22                  }
23          }
24  }
```

First, we need to declare that lifecycle callbacks are present by using the `@HasLifecycleCallbacks` annotation. Then we add the lifecycle annotation `@PrePersist` to the `generatePassword()` method.

That's it! Now, before persisting a new entity, this method is called automatically, and the `User`'s password is auto-generated.

Cascading Operations

When creating a new entity or modifying an existing one, all operations by default affect only a single entity. A powerful, but somewhat dangerous, feature is the option to define "operation cascades." Let's consider the following example. If we delete a `User` in our demo application, we also want all of its `Posts` to be deleted. This can be achieved by adding the `cascade` attribute to the association definition:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @HasLifecycleCallbacks
7   * @Table(name="users")
8   */
9   class User
10  {
11          // [..]
12
13          /**
14          * @OneToMany(targetEntity="Entity\Post", mappedBy="user",
              cascade={"remove"})
15          */
16          private $posts;
17
18          // [..]
19  }
```

Now, when removing a user via

```
<?php
$user = $em->getRepository('Entity\User')->find($id);
$em->remove($user);
$em->flush();
```

the User is gone, and so are all its Posts. When setting cascade to value all, the cascade will be applied on other operations, such as persist, as well.

When adding the cascade attribute, the side matters. In the code shown above, all referenced Post entities are removed when a User is removed. When adding cascade to the Post entity as shown in the following code, the User entity will be removed if one of its referenced Post entities is deleted:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
*/
class Post
{
        // [..]

        /**
        * @ManyToOne(targetEntity="Entity\User", inversedBy="posts",
          cascade={"remove"})
        * @JoinColumn(name="user_id", referencedColumnName="id")
        */
        protected $user;

        // [..]
}
```

Assuming that this is not the desired behavior, it is absolutely crucial to verify the cascade configuration to prevent data loss.

Another way to achieve automatic deletion of referenced entities as shown above is *orphan removal* for one-to-one and one-to-many associations. Orphan removal means Doctrine 2 will automatically remove referenced entities without a parent entity:

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

```

```
10          /**
11          * @OneToMany(targetEntity="Entity\Post", mappedBy="user",
              orphanRemoval=true)
12          private $posts;
13
14          // [..]
15  }
```

Again, when the `User` is gone, all of its `Posts` are also gone.

While cascading operation are useful, they can be expensive. The reason is that all operations on the referenced entities happen in RAM. The entities must first be loaded and reconstructed from the database, and then modified. Depending on the size of the collection, this could be resource intensive.

Luckily, Doctrine 2 also offers "database level" cascading operations for updates and deletes via the `@JoinColumn` annotation:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity(repositoryClass="Repository\Post")
6   * @Table(name="posts")
7   */
8   class Post
9   {
10          /**
11          * @ManyToOne(targetEntity="Entity\User", inversedBy="posts")
12          * @JoinColumn(name="user_id", referencedColumnName="id",
              onDelete="CASCADE")
13          */
14          protected $user;
15
16          // [..]
17  }
```

Transactions

A transaction is an atomic unit of one or more database statements. All insert, update, or delete operations done through the entity manager are queued, as long as the `flush()` method has been called on the entity manager. Technically speaking, the queue is an implementation of the so-called *unit of work*[1] pattern. When calling `flush()`, all queued operations in the unit of work are fired against the database as a single transaction. If one of these operations fails Doctrine 2 automatically rolls back the entire transaction—that is, all operations queued—and then quits, itself, to prevent data loss due to inconsistencies.

[1] http://martinfowler.com/eaaCatalog/unitOfWork.html

Doctrine 2 offers a convenient way to wrap several database operations into a single transaction. The following code demonstrates how to "reset" a `User` in the demo application. First, the existing `User` is deleted, then a new one is created by using its current first and last name:

```
<?php
// [..]
$em->transactional(function($em) {
        $oldUser = $em->getRepository('Entity\User')->find(1);
        $newUser = new Entity\User();
        $newUser->setFirstName($oldUser->getFirstname());
        $newUser->setLastName($oldUser->getLastname());
        $em->persist($newUser);
        $em->remove($oldUser);
});
```

Both operations take effect only if no exception was thrown for either of them. Otherwise, both operations are rolled back.

Another issue may arise when two or more persons simultaneously work on the same sets of data, which is not unlikely for web applications. Doctrine 2 fully supports a strategy called *optimistic locking*. The core idea behind optimistic locking is that multiple users can all read data sets, however, whenever changing data, only the first person writing is privileged to persist its changes. All other users, now working with an outdated version of the entity, will get an exception when trying to persist their individual changes. As this strategy allows for concurrent reading operations and controls only write operations, read-intensive applications won't be slowed down, compared to a pessimistic locking strategy.

To make optimistic locking happen, Doctrine 2 allows us to add a special `integer` or `datetime` version field to an entity. The current value of this field is compared to the value loaded before, and if it doesn't match on write, an `OptimisticLockException` is thrown. If this happens, another user must have already modified the entity. A version field can be set up by adding the `@Version` annotation:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
*/
class Post
{
        // [..]

        /** @Version @Column(type="integer") */
        protected $version;

        // [..]
}
```

When creating the corresponding data schema via Doctrine 2, a column called `version` is added to the `posts` table. For every new `Post` entity, Doctrine 2 automatically assigns a value of `1`. The value is increased by one with each subsequent modification:

```
1   $post = $em->getRepository('Entity\Post')->find(1);
2   $post->setTitle("New title");
3   $em->flush();
4   $em->clear();
5   $post = $em->getRepository('Entity\Post')->find(1);
6   $post->setTitle("Again, a new title");
7   $em->flush();
```

The code shown above modifies the `Post` entity two times with an ID value of `1`. After both changes have been applied, the `value` column has a value of `3`. The `$em->clear()` statement is important here: if we omit it, we would produce an `OptimisticLockException` ourselves. The reason is that changing the title to "New title" and flushing the entity manager increments the version value for this entity by one in the database. However, the value stored in the `Post` entity object in RAM still has the old value of `1`, and therefore is now outdated—it is not being updated to the latest version value automatically. Therefore, trying to modifying the title again will fail with an `OptimisticLockException`. The same will be true if we try to persist changes to an entity which has been modified in the meantime by somebody else. How an `OptimisticLockException` situation is handled is fully up to the application developer.

Summary

In this chapter, we got our hands dirty creating and manipulating entities programatically. We also looked at the entity lifecycle and cascading features making our code even more compact, but also introduces some „magic“ and should be used with care. Transactions are another major aspect of data persistence and is covered well by Doctrine. For the most part, Doctrine already takes care of proper transaction demarcation for you: All the write operations are queued until EntityManager#flush() is invoked which wraps all of these changes in a single transaction. However, Doctrine also allows (and encourages) you to take over and control transaction demarcation yourself.

CHAPTER 8

Doctrine Query Language

Introduction

Earlier in this book, we learned about repositories, containers for entities of a specific type. They are used to look up entities by specific criteria, do updates or deletes, and so on. They do their work with the help of finder methods, which are implemented using Doctrine's own entity query language DQL, the Doctrine Query Language. Strictly speaking, DQL is not bound to finder methods or repositories; however, it is usually good practice to put all DQL statements there, just to keep things organized.

DQL itself is a language to query entities. It looks much like SQL, which makes learning DQL easier, but it isn't SQL. While DQL statements can be written as a string like this

```
1   <?php
2   // [..]
3   $query = $em->createQuery(
4            'SELECT u FROM Entity\User u WHERE u.lastName = "Mustermann"'
5   );
6
7   $users = $query->getResult();
```

it is much more convenient to use the *query builder*, especially when constructing dynamic queries:

```
 1   <?php
 2   // [..]
 3   $qb = $this->_em->createQueryBuilder();
 4
 5   $qb->select('u')
 6           ->from('Entity\User', 'u')
 7           ->where($qb->expr()->eq('u.lastName', '?1'))
 8           ->setParameter(1, "Mustermann");
 9
10   $users = $qb->getQuery()->getResult();
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_8

We assume that `$this->_em` holds a reference to the entity manager (which is true for every repository that extends `Doctrine\ORM\EntityRepository`). The entity manager is capable of providing a query builder, which in turn can be used to programmatically construct a query. Thanks to its fluent interface, the code looks pretty elegant. In contrast to the first example, we also utilize value parameters and the `Expr` class, which we will look at in detail in a minute.

Retrieving Results

When executing a query, multiple options exist for retrieving results. When calling `getResult()` on a query object, a PHP array is returned containing all matching entity objects. Alternatively, `getArrayResult()` can be used to get all data in the form of an array. No objects are returned, only all entities' data as an array in a container array. This is useful when dealing with large datasets or for simple display tasks, where no objects are needed in the processing. The method `getScalarResult()` returns a similar result, but fully flat, not nested at all. When a single result is desired, calling `getSingleResult()` or `getSingleScalarResult()` will do. Method `getOneOrNullResult()` may be used if null is desired when no match was found.

If a query includes objects as well as scalar values as well, the result set returned is called "mixed":

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('u')
        ->addSelect($qb->expr()->concat('u.firstName', 'u.lastName'))
        ->from('Entity\User', 'u')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

In this query, we not only retrieve entities, but also concatenate the user's first and last names. The result of the query looks like this:

```
array
        [0]
              [0] => Object
              [1] => "Max Mustermann"
        [1]
              // ..
```

The result set can be limited, via `setFirstResult($offset)` and `setMaxResults($limit)`, as when building a pagination feature.

Another feature Doctrine 2 offers is retrieving partial objects, entities which have been only partially recreated from the database. To retrieve a partial object, a special syntax is required:

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('partial u.{id, firstName}')
        ->from('Entity\User', 'u')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

This way, we get back partly reconstituted `User` objects from the database. When omitting the `partial` syntax and simply stating individual fields, the result is a plain array without objects:

```
array(1) {
        [0]=>
                array(2) {
                        ["id"]=> int(1)
                        ["firstName"]=> string(3) "Max"
                }
}
```

It's important to always include the identifier (`id` in this case) in a partial object definition. Otherwise an exception is thrown.

While Partial objects can be very helpful, such as while tweaking the performance of an app, they can be problematic as well. Code dealing with partial objects needs to be aware of the fact that no "real" entities are returned, and certain fields or associations might not be available. Use partial objects with care.

Constructing Basic Queries

The query builder provides methods for the different parts of a query, such as the one shown above:

- public function select($select = null);
- public function delete($delete = null, $alias = null);
- public function update($update = null, $alias = null);
- public function set($key, $value);
- public function from($from, $alias = null);
- public function where($where);
- public function andWhere($where);
- public function orWhere($where);
- public function groupBy($groupBy);

- public function addGroupBy($groupBy);
- public function having($having);
- public function andHaving($having);
- public function orHaving($having);
- public function orderBy($sort, $order = null);
- public function addOrderBy($sort, $order = null);

Expressions like the ones shown above are built using an `Expr` object, which is provided by the query builder when calling its `expr()` method. The `Expr` object provides several methods with which to construct an expression:

- public function andX($x = null);
- public function orX($x = null);
- public function eq($x, $y);
- public function neq($x, $y);
- public function lt($x, $y);
- public function lte($x, $y);
- public function gt($x, $y);
- public function gte($x, $y);
- public function isNull($x);
- public function isNotNull($x);
- public function prod($x, $y);
- public function diff($x, $y);
- public function sum($x, $y);
- public function quot($x, $y);
- public function exists($subquery);
- public function all($subquery);
- public function some($subquery);
- public function any($subquery);
- public function not($restriction);
- public function in($x, $y);
- public function notIn($x, $y);
- public function like($x, $y);
- public function between($val, $x, $y);

- public function trim($x);
- public function concat($x, $y);
- public function lower($x);
- public function upper($x);
- public function length($x);
- public function avg($x);
- public function max ($x);
- public function substr($x, $from, $len);
- public function min($x);
- public function abs($x);
- public function sqrt($x);
- public function count($x);
- public function countDistinct($x);

Expressions are used in the `SELECT`, `WHERE`, `HAVING` or `GROUP` part of a query. However, the query shown above can also be created without using the `Expr` class:

```
1   <?php
2   $qb = $this->_em->createQueryBuilder();
3
4   $qb->select('u')
5           ->addSelect("CONCAT(u.firstName, u.lastName)")
6           ->from('Entity\User', 'u')
7           ->where('u.lastName = ?1')
8           ->setParameter(1, "Mustermann");
9
10  $users = $qb->getQuery()->getResult();
```

This might be necessary sometimes, since not every function or arithmetic operator can be constructed via the `Expr` class. The following aggregate functions are allowed in `SELECT` and `GROUP BY` clauses:

- AVG
- COUNT
- MIN
- MAX
- SUM

The following functions are supported in `SELECT`, `WHERE`, and `HAVING` clauses:

- IDENTITY
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE()
- CURRENT_TIME()
- CURRENT_TIMESTAMP()
- LENGTH(str)
- LOCATE(needle, haystack [, offset])
- LOWER(str)
- MOD(a, b)
- SIZE(collection)
- SQRT(q)
- SUBSTRING(str, start [, length])
- TRIM([LEADING | TRAILING | BOTH] [“trchar” FROM] str)
- UPPER(str)
- DATE_ADD(date, days, unit)
- DATE_SUB(date, days, unit)
- DATE_DIFF(date1, date2)

Values can be given for placeholders within queries via `setParameter()` or `setParameters($array)`. In the example shown above, number placeholders are used (starting with a “?” symbol). Alternatively, a string placeholder may be used (starting with a “:” symbol):

```
1   <?php
2   // [..]
3   $qb = $this->_em->createQueryBuilder();
4
5   $qb->select('u')
6           ->addSelect($qb->expr()->concat('u.firstName', 'u.lastName'))
7           ->from('Entity\User', 'u')
8           ->where($qb->expr()->eq('u.lastName', ':lastName'))
9           ->setParameter("lastName", "Mustermann");
10
11    $users = $qb->getQuery()->getResult();
```

Whichever way is preferred, one needs to stick to it within a query. Mixing is not allowed.

Constructing Join Queries

Doctrine 2 supports two different types of joins. While regular joins are needed, for example, to limit results via a `WHERE` clause, so-called *fetch joins* are used to fetch related entities for further usage:

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('u', 'c')
        ->from('Entity\User', 'u')
        ->leftJoin('u.contactData', 'c')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

The `select('u', 'c')` makes the join a fetch join. The referenced `ContactData` object is part of the result set. However, when omitting the `'c'` in the `select` clause, the `ContactData` object is not part of the result anymore, however, it can still be used, for example, within the `where` clause:

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

$qb = $this->_em->createQueryBuilder();

$qb->select('u')
        ->from('Entity\User', 'u')
        ->leftJoin('u.contactData', 'c')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->andWhere($qb->expr()->eq('c.email', '?2'))
        ->setParameter(1, "Mustermann")
        ->setParameter(2, "max.mustermann@example.com");

$users = $qb->getQuery()->getResult();
```

Summary

We covered the most fundamental aspects of Doctrine's very own query language. As said earlier, DQL itself is a language to query entities. It looks much like SQL-which makes learning DQL easier- but it isn't SQL. When working with Doctrine, a proper understanding of DQL is needed.

CHAPTER 9

Command Line Tools

Doctrine 2 ships with powerful command line support. The command line tools can broadly be divided into those that support a database abstraction layer (DBAL-related operations) and those that provide object-relational mapping (ORM-related operations). A list of all commands available can be printed via the `list` command:

```
1   $ ./doctrine list
```

The `help` command, or the command parameter `--help` in conjunction with other commands, prints the instructions for a given command. For example, if we need help dealing with the `orm:info` command, we can ask for help like this:

```
1   $ ./doctrine orm:info --help
```

This will also work:

```
1   $ ./doctrine help orm:info
```

Both commands print the identical output to the console.

Setting Up the Command Line Tools

Before Doctrine 2 can be used on the command line, some basic configuration code is needed. Like the application itself, the command line tools require a ready-to-go database connection and entity manager if ORM-related commands are needed.

The configuration of the command line tools first looks somewhat strange. Doctrine 2 requires that a file called `cli-config.php` exists within the folder in which the tools are executed on the command line. If Doctrine 2 was installed using Composer, the command line tools are located in `vendor/bin`. Therefore, this is the place where a file called `cli-config.php` needs to be set up with the following basic code:

```
1   <?php
2   include '../../vendor/autoload.php';
3
4   use Doctrine\ORM\Tools\Setup;
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_9

```
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);

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper(
        $em->getConnection()
),
'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper(
        $em
)
));
```

The code shown above is similar to the `index.php` file used in the demo app. First, autoloading is configured, then the entity manager is instantiated. If both clients, the web application and the command line tools, share the same credentials, externalizing this data may be helpful.

When the command line tools are invoked, Doctrine 2 automatically includes the `cli-config.php` file and also looks for a so-called *helper set* defined in the global namespace. The *helper set* provides the command line tools with the database connection via key `db` (needed for the DBAL commands) as well as the entity manager via key `em` (needed for the ORM commands).

DBAL Commands

Execute an SQL Statement

SQL commands can easily be executed via the command line with Doctrine's command line tooling. The command `dbal:run-sql` requires a single parameter, a valid SQL statement:

```
$ ./doctrine dbal:run-sql "SELECT * FROM users;"
```

The result of the query is printed to the console.

Import SQL Files

If you need to execute multiple statements, importing an SQL file via the `./doctrine dbal:import` command might be a great option. It takes one or more paths to SQL files, delimited by space:

```
1   $ ./doctrine dbal:import /tmp/import-data.sql
```

ORM Commands

Validate Persistence Configuration

One of the most helpful commands is `orm:validate-schema`, which validates the current persistence configuration and makes sure that it matches the existing data schema in the database. If all is good, the following command prints a positive message to the console:

```
1   $ ./doctrine orm:validate-schema
2   > [Mapping] OK - The mapping files are correct.
3   > [Database] OK - The database schema is in sync with the mapping files.
```

The command `orm:info` prints an overview of the application entities known to Doctrine:

```
1   $ ./doctrine orm:info
2   > Found 2 mapped entities:
3   > [OK] Entity\Post
4   > [OK] Entity\User
```

The Schema Tool

With the help of the commands

1. `orm:schema-tool:create`,
2. `orm:schema-tool:drop`
3. `orm:schema-tool:update`

one can manipulate a database data schema based on the entity persistence configuration. However, when running these commands, nothing actually happens—it's just a "dry run". If the commands are executed with parameter `--dump-sql`, again, only a dry run occurs. However, this time, the schema tool prints all SQL statements, which would otherwise have been fired against the database, to the screen. Only if one uses the parameter `--force`, does the schema tool finally execute the statements:

```
1   $ ./doctrine orm:schema-tool:drop --force
```

The command shown above makes all tables and data disappear:

```
1   > Dropping database schema...
2   > Database schema dropped successfully!
```

 **Danger! Potential loss of data!** The schema tool can delete tables and/or data. Use these commands with caution.

The execution of

```
1   $ ./doctrine orm:schema-tool:create
```

creates the data structure from scratch, while

```
1   $ ./doctrine orm:schema-tool:update
```

migrates an existing data schema from status quo to match the current persistence configuration, if the existing data schema is not yet up-to-date.

Generate Commands

With the help of the command `orm:generate-proxies`, the proxy classes for the entities defined can be created, which otherwise are created by Doctrine 2 automatically when needed. Via the command `orm:generate-entities`, the entity classes themselves can be generated; this is especially helpful if the persistence configuration is given via XML or YAML. Executing the command

```
1   $ ./doctrine orm:generate-entities /tmp
```

creates the entity classes in `/tmp`, or to be precise, in `/tmp/Entity`. More options for the command `orm:generate-entities` can be identified running the following command:

```
1   $ ./doctrine orm:generate-entities --help
```

With the help of the command `orm:generate-repositories`, repository classes can also be auto-generated.

Execute a DQL Command

Similar to `dbal:run-sql`, DQL statements can also easily be executed using the command `orm:run-dql`.

Cache-Related Commands

With the commands

1. `orm:clear-cache:metadata`,
2. `orm:clear-cache:query`
3. `orm:clear-cache:result`

the various Doctrine caches can be purged. These commands are especially helpful when deploying a new application version. They make sure that no outdated configuration or data is used in conjunction with a new application version—usually, this would lead to unrecoverable errors.

Converting Commands

Rarely used, but very helpful, are the commands `orm:convert-d1-schema` and `orm:convert-mapping`. The command `orm:convert-d1-schema` is used to transform the old Doctrine 1 persistence configuration format to the one used by Doctrine 2, which is very handy when upgrading an existing application. The command `orm:convert-mapping`, on the other hand, allows you to go from one Doctrine 2 mapping format to another, such as from XML to YAML.

Production-Ready Configuration

The following command validates that Doctrine's configuration is ready for production usage:

```
1   $ ./doctrine orm:ensure-production-settings
```

If this is the case, it prints a positive message on the console:

```
1   > Environment is correctly configured for production
```

Many configuration aspects can cause a negative result. This could be missing proxy classes created by running the proper command or missing caches. Both are considered by Doctrine 2 to be a requirement for usage in productive systems.

Custom commands Doctrine 2 allows you to develop custom commands that can be hooked into the command line tooling. More information about this topic can be found in the official documentation.[1]

[1] http://docs.doctrine-project.org/en/latest/reference/tools.html#adding-own-commands

Summary

The Doctrine CLI tools are extremely helpful. They can be used in shell or build scripts and also serve well when importing external data such as test fixtures or dealing with database schema changes. Since the CLI tools are also extendable, they are very versatile and should be part of every developer's toolkit.

CHAPTER 10

Caching

Introduction to ORM Cache Types

Due to its nature and the way Doctrine 2 works, applications using Doctrine 2 naturally run a bit slower than others. However, with good caching strategies applied, this issue can be almost completely eliminated. Doctrine 2 ORM brings native support for three different types of caches: the "meta cache," the "query cache," and the "result cache." When setting up the entity manager, the different caches can be added to the entity manager's configuration. In addition, the cache instance might be used for custom values as well.

Caching Backends

Cached data can be stored in different so-called caching backends. Doctrine 2 supports multiple technologies to be used as caching backends:

- ApcCache (requires ext/apc)
- ArrayCache (in memory, lifetime of the request)
- FilesystemCache (not optimal for high concurrency)
- MemcacheCache (requires ext/memcache)
- MemcachedCache (requires ext/memcached)
- PhpFileCache (not optimal for high concurrency)
- RedisCache.php (requires ext/phpredis)
- WinCacheCache.php (requires ext/wincache)
- XcacheCache.php (requires ext/xcache)
- ZendDataCache.php (requires Zend Server Platform)

Based on the situation or technologies used, you can choose between the different technologies.

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_10

Metadata Cache

The metadata cache holds the entity mapping data given as annotations or external XML or YAML files. Caching this data means that Doctrine 2 does not need to perform reflection or XML or YAML parsing for every single request, which saves a significant amount of processing time.

When setting up the entity manager, the cache can easily be configured:

```
<?php
// [..]
$paths = array(__DIR__ . "/../src/Entity/");
$isDevMode = false;

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

$config = Setup::createAnnotationMetadataConfiguration(
        $paths, $isDevMode
);

$config->setMetadataCacheImpl(
        new \Doctrine\Common\Cache\FilesystemCache('/tmp/doctrine2')
);

$em = EntityManager::create($dbParams, $config);
```

In the configuration shown above, we tell Doctrine 2 to cache metadata locally in the filesystem. We could use other caching backends here as well. The caching path needs to be given when using the `FilesystemCache`. Next time, when entities are processed, Doctrine 2 starts setting up a somewhat cryptic files and folders structure in the given caching path. Luckily, you don't need to care about it—Doctrine 2 takes care of all things caching. We only need to make sure that the caching path given is writable to Doctrine 2. When caching via Memcached or Redis, these services need to be up and running and accessible to Doctrine 2, as well. If not, an exception will be raised.

Query Cache

The query cache ensures that DQL statements need to be translated into SQL only once. This again speeds up a Doctrine 2 application significantly:

```
<?php
// [..]
$paths = array(__DIR__ . "/../src/Entity/");
```

```
4   $isDevMode = false;
5
6   $dbParams = array(
7           'driver' => 'pdo_mysql',
8           'user' => 'root',
9           'password' => '',
10          'dbname' => 'app',
11  );
12
13  $config = Setup::createAnnotationMetadataConfiguration($paths,
    $isDevMode);
14  $cachingBackend = new \Doctrine\Common\Cache\FilesystemCache('/tmp/
    doctrine2');
15  $config->setMetadataCacheImpl($cachingBackend);
16  $config->setQueryCacheImpl($cachingBackend);
17  $em = EntityManager::create($dbParams, $config);
```

In the configuration shown above, we first set up a general caching backend, which now powers both the metadata cache and the query cache.

Result Cache

Last but not least, there is the result cache. The use of a result cache prevents executing the same queries against the database again and again:

```
1   <?php
2   // [..]
3   $paths = array(__DIR__ . "/../src/Entity/");
4   $isDevMode = false;
5
6   $dbParams = array(
7           'driver' => 'pdo_mysql',
8           'user' => 'root',
9           'password' => '',
10          'dbname' => 'app',
11  );
12
13  $config = Setup::createAnnotationMetadataConfiguration($paths,
    $isDevMode);
14  $cachingBackend = new \Doctrine\Common\Cache\FilesystemCache('/tmp/
    doctrine2');
15  $config->setMetadataCacheImpl($cachingBackend);
16  $config->setQueryCacheImpl($cachingBackend);
17  $config->setResultCacheImpl($cachingBackend);
18  $em = EntityManager::create($dbParams, $config);
```

Via method `setResultCacheImpl()`, the result cache now is ready for action. In contrast to the two other caching types, you have to actively tell Doctrine 2 to cache results for a given query:

```
1   <?php
2
3   // [..]
4   $query = $em->createQuery($dqlString);
5   $query->useResultCache(true);
```

Now, the result is cached.

Summary

Since object-relational-mapping results in a noticeable runtime overhead, caching is essential for high speed. In fact, in production, caching is a must and should be setup right from the beginning.

CHAPTER 11

Advanced Topics

Framework Integrations

If one of the popular PHP frameworks is used for an application, integration of Doctrine mostly can be easily done. As an example, we will walk through the process of integrating Doctrine into a Zend Framework 2 application. If you are using a different framework, the official framework documentation or a quick web search usually brings up a suited tutorial.

The easiest way to use to Doctrine 2 with Zend Framework 2 (ZF2) is a Composer based installation. The ZF2 module doctrine-orm-module[1] not only ships the glue code to make both libraries work together, but also ensures that the Doctrine 2 library itself is downloaded and installed in an existing ZF2 application. One simply needs to add the following line to the `require` block of the projects' `composer.json` file:

```
1   "doctrine/doctrine-orm-module": "dev-master"
```

You must also add the following line above the `require` block:

```
1   "minimum-stability": "alpha"
```

If this configuration is missing, Composer might refuse to install the module. In fact, with this configuration, we tell Composer to install even non-stable modules.

The following commands start the download and the installation process:

```
1   $ php composer.phar update
```

As you can see, a whole bunch of other ZF2 modules and additional libraries are downloaded. Last but not least, two ZF2 modules must be activated via the `application.config.php`:

```
1   <?php
2   return array(
3           'modules' => array(
4                   'Application',
```

[1] https://github.com/doctrine/DoctrineORMModule

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_11

```
                'DoctrineModule',
                'DoctrineORMModule'
        ),
        'module_listener_options' => array(
                'config_glob_paths' => array(
                        'config/autoload/{,*.}{global,local}.php',
                ),
                'module_paths' => array(
                        './module',
                        './vendor',
                ),
        ),
);
```

The modules register several services in Main Service Manager, all starting with the label "doctrine," such as `doctrine.cache.apc` and `doctrine.sqlloggercollector.ormdefault`.

The entity manager can be obtained using a long-winded label:

```
<?php
// [..]
$this->getServiceLocator()->get('doctrine.entitymanager.orm_default');
```

But before this will work, some additional configuration is needed. Doctrine 2 needs to know where the entity classes are located and what the caching strategy looks like. The following example from the `module.php` tells Doctrine that the mappings are given as annotation, the entities are located in `__DIR__ . '/../src/Application /Entity'`, and caching takes place via PHP arrays:

```
<?php
// [..]
'doctrine' => array(
        'driver' => array(
                'my_annotation_driver' => array(
                        'class' => 'Doctrine\ORM\Mapping\Driver\
                        AnnotationDriver',
                'cache' => 'array',
                'paths' => array(__DIR__ . '/../src/Application/
                Entity')
                )
        )
)
// [..]
```

To allow other ZF2 modules to provide entities to the application, a so-called Driver Chain allows us to combine multiple entity sources even with different mapping formats and caching strategies:

```
<?php
// [..]
'doctrine' => array(
        'driver' => array(
                'my_annotation_driver' => array(
                        'class' => 'Doctrine\ORM\Mapping\Driver\
                        AnnotationDriver',
                        'cache' => 'array',
                        'paths' => array(__DIR__ . '/../src/
                        Application/Entity')
                ),
                'orm_default' => array(
                        'drivers' => array(
                'Application\Entity' => 'my_annotation_driver'
            )
        )
        )
)
// [..]
```

Now the database connection must be configured. Usually, a dedicated config file in the autoload folder is used, e.g. `db.local.php`:

```
<?php
return array(
         'doctrine' => array(
                 'connection' => array(
                         'orm_default' => array(
                                 'driverClass' => 'Doctrine\DBAL\Driver\
                                 PDOMySql\Driver',
                                 'params' => array(
                                         'host' => 'localhost',
                                         'port' => '3306',
                                         'user' => 'username',
                                         'password' => 'password',
                                         'dbname' => 'database',
                                 )
                         )
                 )
         ),
);
```

Once done, we can start dealing with entities (e.g. a Product entity) through Doctrine 2:

```
<?php
namespace Application\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity
* @ORM\Table(name="product")
*/
class Product
{
        /**
        * @ORM\Id @ORM\Column(type="integer")
        * @ORM\GeneratedValue
        */
        protected $productId;

        /** @ORM\Column(type="string", nullable=true) */
        protected $name;

        /** @ORM\Column(type="integer") */
        protected $stock;

        /** @ORM\Column(type="string", nullable=true) */
        protected $description;

        /** @ORM\Column(type="string", nullable=true) */
        protected $features;

        public function setDescription($description)
        {
                $this->description = $description;
        }

        public function getDescription()
        {
               return $this->description;
        }

        public function setFeatures($features)
        {
                $this->features = $features;
        }

        public function getFeatures()
        {
```

```
47                      return $this->features;
48              }
49
50              public function setProductId($productId)
51              {
52                      $this->productId = $productId;
53              }
54
55              public function getProductId()
56              {
57                      return $this->productId;
58              }
59
60              public function setName($name)
61              {
62                      $this->name = $name;
63              }
64
65              public function getName()
66              {
67                      return $this->name;
68              }
69
70              public function setStock($stock)
71              {
72                      $this->stock = $stock;
73              }
74
75              public function getStock()
76              {
77                      return $this->stock;
78              }
79      }
```

From a controller, a finder method provided by a repository can be accessed like this:

```
1   <?php
2   // [..]
3   $this->getServiceLocator()->get('doctrine.entitymanager.orm_default')
4           ->getRepository('Application\Entity\Product')
5           ->findOneByProductId($id);
6   // [..]
```

The Doctrine 2 command line tools are available as well. You simply bring up a command line and change to the project's root folder:

```
1   $ php vendor/bin/doctrine-module orm:validate-schema
```

Native SQL Statements

Doctrine 2 ships with a special class called `NativeQuery` that allows you to execute native SQL select statements and to map the results returned to entity objects. The same operations that work out-of-the-box with Doctrine 2 can be implemented by hand in cases where native queries are needed or where they are the better solution to a problem. `NativeQuery` allows you to retrieve "raw data" and then subsequently work with entity objects. The official documentation[2] holds further information about how to properly implement native SQL statements.

Lastly, in general, one needs to remember that it's always possible to execute arbitrary SQL statements via the underlying database connection:

```
1   <?php
2   // [..]
3   $em->getConnection()->exec('DELETE FROM posts');
```

While this is possible, it should always be the last resort. It bypasses the Entity Manager and might produce hard-to-debug issues and data inconsistencies.

Doctrine 2 Extensions

While Doctrine 2 already ships with tons and tons of features, there is even more. In the Doctrine 2 extension repository[3] on GitHub, you will find several extensions with solutions to typical problems which otherwise must be solved individually by each application developer again and again:

`Tree`: Automates the tree handling process and adds some tree-specific functions on repositories.

`Translatable`: Gives a very handy solution for translating records into different languages.

`Sluggable`: Takes a specified field from an entity and makes it compatible for URLs.

`Timestampable`: Updates date fields on creates, updates, and even property changes.

`Blameable`: Updates string or reference fields on creates, updates, and even property changes with a string or object (e.g. user).

[2] http://docs.doctrine-project.org/en/latest/reference/native-sql.html
[3] https://github.com/l3pp4rd/DoctrineExtensions

`Loggable`: Helps tracking changes and history of objects, also supports version management.

`Sortable`: Makes any entity sortable.

`Translator`: Supports handling translations.

`Softdeleteable`: Allows you to mark entities as deleted, without physically deleting them.

`Uploadable`: Provides file upload handling to entity fields.

`References`: Supports linking entities in Documents and vice versa.

Summary

Congratulations! You made to it to the end of this book. I thank you very much for buying and reading my book on Doctrine 2 ORM. I believe you now have all knowledge and tools at hand to use Doctrine 2 in your own applications. As in every technical book, we didn't cover all of the features of Doctrine and it might be worth to continue learning by reading the official documentation. The contents of this book will help you to grasp other features and implementation details of Doctrine 2 not covered in this book.

Again, thanks for reading my book and happy coding!

Index

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

O

P

Q

R

S

T, U, V, W, X, Y

Z