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:

```
<?php
namespace Entity;

class User
{
        // [..]

   private $postRepository;

   public function getPosts()
       {
           if (is_null($this->posts)) {
               $this->posts = $this->postRepository->findByUser($this);
           }

           return $this->posts;
       }

       // [..]
}
```

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:

```
<?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;
        }

```

```
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.