Persistenz in PHP mit Doctrine 2 ORM

Grundlagen, Konzepte und praktische Anwendung

Michael Romer

Persistenz in PHP mit Doctrine 2 ORM

Grundlagen, Konzepte und die praktische Anwendung

Michael Romer

Dieses Buch können Sie hier kaufen http://leanpub.com/doctrine2

Diese Version wurde auf 2013-08-23 veröffentlicht

Ebenfalls von Michael Romer

Webentwicklung mit Zend Framework 2

Web Development with Zend Framework 2

PHP Data Persistence with Doctrine 2 ORM

Inhaltsverzeichnis

1 Über dieses Buch (verfügbar)

1.1 Early Access Edition

Wenn du diesen Abschnitt liest, hälst du die "Early Access Edition" dieses Buches in deinen Händen. "Early Access" bedeutet, dass du bereits vorne im Buch mit dem Lesen anfangen kannst, während die weiteren Kapitel noch "in der Mache sind". Sie werden dir automatisch zur Verfügung gestellt, sobald sie fertig sind. Dank' der Idee des Lean Publishing[1] bist du jetzt also noch viel mehr "up-to-date", als du es jemals zuvor warst. Es ist sehr wahrscheinlich, dass du zum aktuellen Zeitpunkt noch eine Reihe von Rechtschreibfehlern und den ein oder anderen Bug in den Code-Beispielen finden wirst. Ich versuche sorgfältig zu arbeiten, kann aber derzeit keine Fehlerfreiheit garantieren. Wenn du mir helfen möchtest, dieses Buch noch besser zu machen, werde doch gerne in der Community zum Buch aktiv und teile dort deine Hinweise und Vorschläge mit. Ich nehme sie dankend entgegen.

1.2 Die Community zum Buch

Du hast einen Fehler gefunden, möchtest deine Ideen für die nächste Auflage loswerden oder dich einfach nur zu den Themen im Buch und zu Doctrine 2 im Allgemeinen austauschen? Dann ist die Community zum Buch[2] bei Google Groups der richtige Anlaufpunkt für dich. Dort bekommst du nicht nur direkten Kontakt zu mir, dem Autor dieses Buches, sondern auch zu den anderen Lesern. Du benötigst lediglich einen Google-Account, den du dir bei Bedarf zuvor kostenlos erstellst.

1.3 Softwareversion

Die Inhalte in diesem Buch beziehen sich auf Doctrine 2.3, sind aber mit hoher Wahrscheinlichkeit auch für spätere Versionen uneingeschränkt gültig.

1.4 Verwendetes Datenbanksystem

Für die Beispiele in diesem Buch kommt das MySQL-Datenbanksystem[3] zum Einsatz. Doctrine 2 unterstützt neben MySQL aber auch weitere DBMS wie etwa PostgreSQL[4] oder SQLite[5]. Der große Teile der Beispiele sind uneingeschränkt auch für diese Systeme gültig, stellenweise müssen ggf. systemspezifische Anpassungen vorgenommen werden.

[1] http://leanpub.com

[2] https://groups.google.com/forum/#!forum/persistenz-in-php-mit-doctrine-2-orm

[3] mysql.de

[4] http://www.postgresql.org/

[5] http://www.sqlite.org/

1.5 Konventionen

Listings werden im gesamten Buch hervorgehoben und sind mit Zeilennummern versehen. An entsprechenden Stellen wird auf ein öffentlich zugängliches Code-Repository bei github[6] verwiesen, über das die Codebeispiele heruntergeladen werden können. Bei Listings, die ein $ vorangestellt haben, handelt es sich um Befehle, die auf der Kommandozeile ausgeführt werden müssen. Auf Kommandos folgende Zeilen, die ein > vorangestellt haben, verstehen Sie sich als Kommandozeilenausgaben nach Kommandos. Neu eingeführte Fachbegriffe werden in *kursiver Schrift* dargestellt. Eine Erklärung dieser Begriffe befindet sich im Glossar im Anhang zum Buch.

1.6 Wichtige Hinweise für Amazon-Kunden

Wenn dieses Buch bei Amazon erworben wurde, ist es derzeit schwieriger, Updates des Buches automatisch bereitzustellen. Um Probleme zu vermeiden, sende mir bitte nach dem Kauf eine kurze E-Mail mit dem Betreff “Updates Doctrine 2” an mail@michael-romer.de. So kannst du sicher sein, dass du tatsächlich immer den aktuellen Stand dieses Buches vorliegen hast. Als kleines Dankeschön für deine Mühen bekommst du zusätzlich dann auch die PDF- und EPUB-Versionen des Buches.

1.7 Weitere Bücher des Autors

Neben diesem Buch ist mit Webentwicklung mit Zend Framework 2[7] ein weiteres Buch des Autors verfügbar. Für alle, die sich für professionelle Webentwicklung interessieren, stellt dieses Buch eine gute Ergänzung dar.

[6] https://github.com/

[7] https://leanpub.com/zendframework2

2 Einleitung (verfügbar)

2.1 Objektorientierung & Domain Models

Als PHP-Entwickler denkt und programmiert man in diesen Tagen in aller Regel objektorientiert. Funktionalität wird über Klassen, Objekte, Methoden, Vererbung und den vielen weiteren Ansätzen und Möglichkeiten der Objektorientierung realisiert. In seiner Anfangszeit hat Objektorientierung in PHP vor allem auf die technischen Aspekte von Applikationen Anwendung gefunden, etwa im Rahmen von *MVC*-basierten Frameworks, Logging- oder Mailing-Bibliotheken. Diese Komponenten verstehen sich aus fachlicher Sicht als allgemeingültige Lösungen, die genauso gut etwa in E-Commerce-Anwendungen, Portallösungen oder Community-Systemen einsetzbar sind. Für komplexere Anwendungen oder grundsätzlich auch immer dann, wenn Systeme nachhaltig änderbar und erweiterbar sein müssen, hat sich die Objektorientierung zudem mittlerweile aber auch in der "Fachlichkeit" einer Anwendung manifestiert.

Grundsätzlich besteht jede Anwendung aus zwei Arten von Code: technisch-orientiertem Code und fachlich-orientiertem Code. Während technisch-orientierter Code in aller Regel eben diesen allgemein gültigen Charakter besitzt und sich gut in Form von Bibliotheken oder Frameworks kapseln, wiederverwenden und verteilen lässt, trifft das auf fachlich-orientierten Code aufgrund seiner Spezialisierung in der Regel nur selten, oft gar nicht zu. Objektorientierter, fachlicher Code zeichnet sich technisch durch das Vorhandensein eines sog. "Domain Models" in der Anwendung aus. Das Domain Model hat meist u.a. die folgenden Charakteristika:

- Es existieren einzelne Klassen/Objekte für die Kernelemente der fachlichen Domäne, die sog. *Entities* (streng genommen werden kann es sich bei diesen Elementen auch um sog. *Value Objects* handeln, dazu aber später mehr). In einem Shopsystem wären dies etwa Klassen wie "Customer", "Order", "Product", "Cart" und so weiter.
- Die fachlichen Klassen/Objekte haben bei Bedarf untereinander Assoziationen, so hätte in einem Shopsystem eine "Order" vermutlich mindestens eine Assoziation zum "Customer" als auch zu den bestellten "Products".
- Fachliche Funktionen sind Teil der jeweiligen Entity. In einem Shopsystem verfügt die "Cart" etwa oft über eine `calculateTotalPrice()`-Methode, die den insgesamt vom Kunden zu bezahlenden Preis auf Basis der im Warenkorb abgelegten Produkte und dessen Anzahl berechnet.
- Funktionen, die sich auf mehrere Entity-Typen beziehen, werden in sog. *Services* implementiert, eben weil sie sich nicht eindeutig einer Entity zuordnen lassen. In einem Shopsystem wäre das etwa der "Checkout", der Warenbestände reduzieren, Rechnungen generieren und Bestellhistorien modifizieren muss, es also mit einer ganzen Reihe von Entities zu tun hat.

- Innerhalb der Anwendung wird wann immer möglich mit den fachlichen Elementen gearbeitet, anstelle von generischen Datencontainern wie etwa Arrays (sinnvolle Ausnahmen bestätigen hier die Regel).
- Geschäftslogik (z.B. auch Geschäftsregeln) wird wann immer möglich innerhalb der fachlichen Klassen implementiert und nicht etwa in den *Controllern* einer Anwendung.

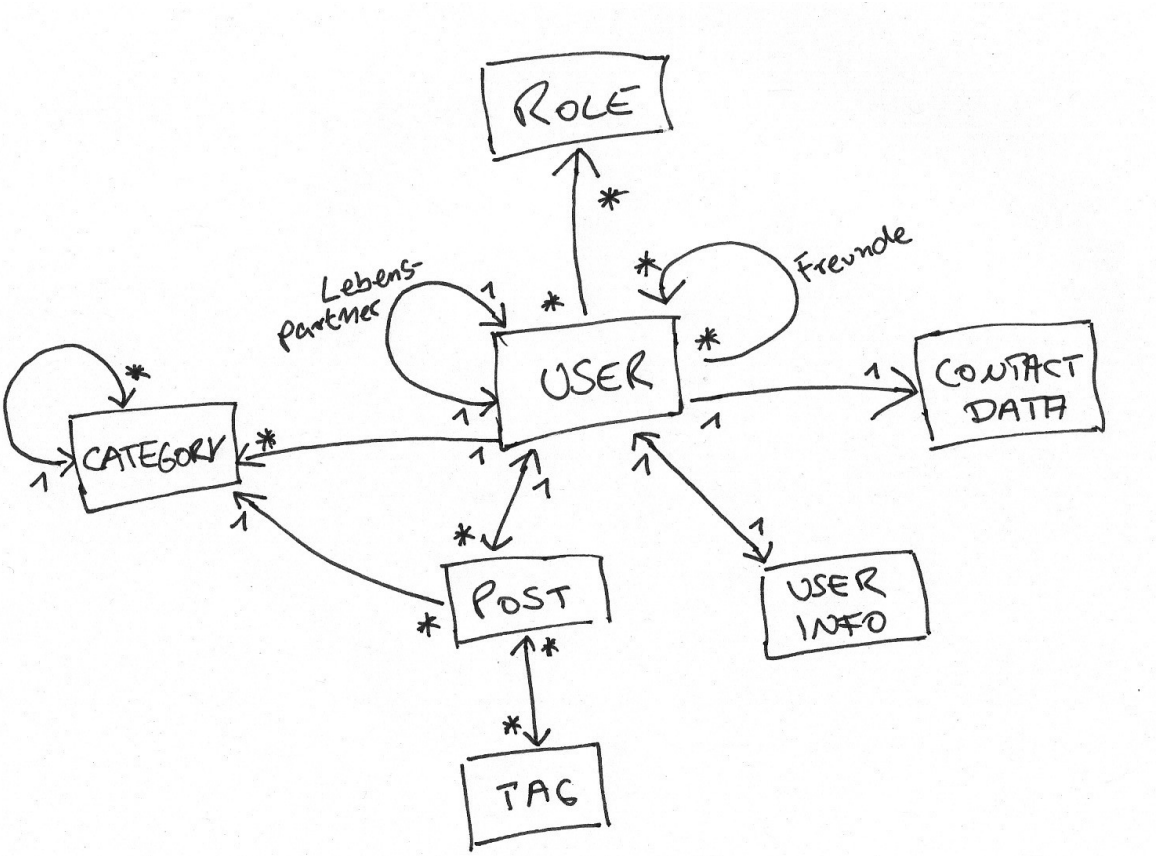
Der große Vorteil des Domain Models liegt in der Kapselung der Fachlichkeit und damit der Unterstützung der Änderbarkeit und Erweiterbarkeit der Anwendung. Die Wahrscheinlichkeit, bei Änderungen versehentlich etwas An ders kaputt zu machen, sinkt. Durch die Isolierung des fachlichen Codes vom technischen Code steigt zudem die Portabilität des Systems. Das ist etwa immer dann sehr hilfreich, wenn man von einem "Application Framework" zu einem anderen wechseln will (oder muss).

Ein Domain Model hat neben der technischen Komponente aber auch weitere Vorteile jenseits des Codes: Wenn an der (Weiter-)Entwicklung eines Produktes nicht nur Programmierer beteiligt sind, sondern auch BWLer, Vertriebler, Marketeers und so weiter - und das ist bei Anwendungen mit wirtschaftlichem Interesse ja fast immer der Fall - ermöglicht es das Domain Model, eine gemeinsame Sprache für die wichtigsten Aspekte der Geschäftstätigkeit zu entwickeln. Diskussionen rund um das gemeinsame Produkt können so deutlich effizienter und zielführender gestaltet werden. Dies ist ein unschätzbarer Vorteil, der für sich alleine genommen fast schon ein Domain Model in der Anwendung rechtfertigt.

2.2 Beispielanwendung

Am besten lernt man an konkreten Beispielen. Auch in diesem Buch hilft uns die immer wiederkehrende Beispielanwendung *Talking* dabei, Theorie und Praxis bestmöglich zu verzahnen. Talking ist eine Webanwendung und ermöglicht es Nutzern, ganz ähnlich zu Wordpress[1] oder Tumblr[2], Texte in Form sog. *Posts* zu veröffentlichen. Das Domain Model von Talking stellt sich wie folgt dar (in den nachfolgenden Ausführungen sind die Entities **durch Fettschrift hervorgehoben**):

[1] https://wordpress.com/
[2] www.tumblr.com



Beispielanwendung "Talking" - Domain Model

Ein **User** kann mehrere **Post**s schreiben, wobei ein **Post** immer nur einen User als Autor haben kann und man sowohl vom User auf dessen Posts, als auch von einem Post auf dessen Autor, einen User, schließen kann. Ein User kann eine oder mehrere **Role**s im System einnehmen. Über einen User lässt sich auf dessen Roles schließen, nicht aber von einer Role-Entity auf die User, die ebenjene Rolle einnehmen. Zu jedem User gibt es eine **UserInfo**, die Informationen über das Datum der Registrierung bei Talking, sowie gegebenenfalls auch das Datum der Abmeldung enthält. Es lässt sich sowohl von einem User auf dessen UserInfo schließen, als bei Bedarf auch andersherum von einer UserInfo auf einen User. Weiterhin verfügt ein User über **ContactData**. Dort wird die E-Mail und Telefonnummer (zusammen in einem "Container") eines Users gespeichert, falls bekannt. Von einem User lässt sich auf dessen ContactData schließen, nicht aber umgekehrt. Weiterhin kann ein User einen anderen User als Lebenspartner kennzeichnen, der sich jeweils von beiden Seiten aus ermitteln lässt, als auch eine unbegrenzte Menge von Usern als Freunde. Man kann von einem User auf dessen Freunde schließen, nicht aber umgekehrt. Ein einzelner Post eines Users kann über beliebig viele **Tag**s (Schlagworte) verfügen und ein Tag kann in mehreren Posts verwendet werden. Sowohl kann von einem Post auf dessen Tags, als auch von einem Tag auf alle Posts, die über dieses Schlagwort verfügen, geschlossen werden. Zusätzlich können mehrere Posts in einer **Category** gruppiert werden. Von einem Post aus lässt sich dessen Category ermitteln, es lassen

sich aber von einer Category aus nicht die dort gruppierten Posts ermitteln. Eine Category kann über untergeordnete Categories verfügen, wobei jeweils von einer Category ggf. auf untergeordnete Categories, als auch eine übergeordnete Category schließen lässt. Die Categories selbst sind einem User zugeordnet, über den auf dessen Categories geschlossen werden kann, nicht aber über eine Kategorie auf dessen User.

Das Domain Model ist so gewählt, dass sich möglichst viele Funktionen von Doctrine 2 auch im Rahmen der Beispielanwendung demonstrieren lassen. In einem produktiven System würde man vermutlich den ein oder anderen Aspekt in einer anderen Form realisieren wollen.

Die oben genannten Einschränkungen beim Zugriff von einer Entity auf andere, referenzierte Entities, sind, wie wir später noch ausführlich sehen werden, Funktionen eines ORM-Systems und basieren auf fundamentalen Gegebenheit der objektorientierten Programmierung. Sie sind hingegen nicht innerhalb einer relationalen Datenbank vorhanden. Doch mehr dazu später.

3 Ein einfaches ORM-System selbstgebaut (verfügbar)

Direkt mit Doctrine 2 starten?

Der folgende Abschnitt verdeutlicht, warum Doctrine 2 so eine große Hilfe für den Anwendungsentwickler ist. Stück für Stück werden Funktionen, die die Bibliothek schon automatisch an Bord hat, exemplarisch "von Hand entwickelt". Bei Bedarf kann dieser Abschnitt übersprungen werden.

3.1 Laden einer Entity

Ein Domain Model ist also eine gute Sache. Und solange man sich als Anwendungsentwickler in der "objektorientierten Welt" aufhält, gibt es auch keine unlösbaren Herausforderungen beim Design und der Implementierung des jeweiligen Domain Models. Schwieriger wird es, wenn es darum geht, nicht mehr nur mit den flüchtigen Daten zu arbeiten, sondern die Daten des Domain Models in relationalen Datenbanken zu speichern, bzw. Objektbäume aus zuvor gespeicherten Daten wieder aufzubauen. Da wäre etwa die Tatsache, dass es sich bei Objekten im Gegensatz zu Datenbanktabellen nicht nur um reine "Datenstrukturen" handelt, sondern Objekte auch über ein definiertes Verhalten in Form von Methoden verfügen können. Nehmen wir das konkrete Beispiel des User-Objekts der Beispielanwendung Talking:

```
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
15      const GENDER_MALE_DISPLAY_VALUE = "Mr.";
16      const GENDER_FEMALE_DISPLAY_VALUE = "Mrs.";
17
18      public function assembleDisplayName()
19      {
20          $displayName = '';
21
22          if ($this->gender == self::GENDER_MALE) {
23              $displayName .= self::GENDER_MALE_DISPLAY_VALUE;
24          } elseif ($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 $displayName;
35      }
36
37      public function setFirstName($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

Interessant ist hier insbesondere die Methode `assembleDisplayName()`, die auf Basis der Daten des Objekts den "Anzeigenamen" für einen User erzeugt, den man z.B. benötigt, wenn man den Autor eines Posts anzeigen will. Der Aufruf von

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

$user = new Entity\User();
$user->setFirstName('Max');
$user->setLastName('Mustermann');
$user->setGender(0);
$user->setNamePrefix('Prof. Dr');

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

Listing 1.2

etwa führt zur Ausgabe von

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

Die Methode `assembleDisplayName()` definiert demnach ein bestimmtes Verhalten des User-Objekts.

Wenn also etwa Daten aus der Datenbank geladen werden, z.B. die Stammdaten eines bereits registrierten Users, müssen sie derart verarbeitet werden, dass sie um das oben skizzierte Verhalten, sprich um die `assembleDisplayName()`- Methode, angereichert werden. Oder anders gesagt: Die Daten eines Users, die aus der Datenbank geladen werden, müssen irgendwie zur weiteren Verarbeitung durch die Anwendung in ein User-Objekt überführt werden, denn in der Anwendung wollen wir ja möglichst immer mit den fachlichen Objekten hantieren um einfach (und konsistent) auf die `assembleDisplayName()`-Methode zugreifen zu können.

Erstellen wir dazu zunächst eine einfache Datenstruktur in der Datenbank:

```sql
CREATE TABLE users(
        id int(10) NOT NULL auto_increment,
        first_name varchar(50) NOT NULL,
        last_name varchar(50) NOT NULL,
        gender ENUM('0','1') NOT NULL,
        name_prefix varchar(50) NOT NULL,
        PRIMARY KEY (id)
);
```

Und fügen die Testdaten für “Max Mustermann” hinzu:

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

Wenn wir nun das User-Objekt auf Basis der zuvor erstellten Daten aus der Datenbank erzeugen wollen, könnten wir dies wie folgt erledigen:

```
<?php
include('../entity/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();
$user->setId($userData['id']);
$user->setFirstName($userData['first_name']);
$user->setLastName($userData['last_name']);
$user->setGender($userData['gender']);
$user->setNamePrefix($userData['name_prefix']);

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

Verbindungsdaten in Produktivsystemen

In einem produktiven Systemen ist es selbstverständlich ratsam, im Gegensatz zum Beispiel oben, mit einem (starken) Passwort und im Idealfall nicht direkt mit dem Nutzer “root” zu arbeiten.

Mit diesen Zeilen haben wir damit begonnen, ein eigenes ORM-System zu schreiben, das hier die Daten einer Tabelle der Datenbank auf die Eigenschaften eines Objekts abbildet. Machen wir damit mal ein wenig weiter.

Um die Logik für ebenjenes “Mapping” zu kapseln, überführen wir den Code in eine eigene Klasse:

```php
<?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;
        }
}
```

So könnte zumindest mal eine rudimentäre Implementierung ausschauen. Wasserdicht ist sie natürlich nicht, aber sie erfüllt ihren Zweck. Der eigentliche Programmablauf sieht jetzt dann so aus:

```php
<?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();
```

Wir können aber noch weiter vereinfachen, wenn wir das SQL-Statement samt den weiteren Arbeitsschritten zur Erzeugung des User-Objekts in einem eigenen Objekt, einem sog. *Repository*, kapseln:

```php
<?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();
```

```
26
27                  return $this->mapper->populate($userData, new UserEntity());
28          }
29  }
```

Nun noch die Datenbankverbindungslogik im *Entity Manager* bündeln und das zuvor erstellte User-Repository darüber bereitstellen:

```
1   <?php
2
3   include_once('../repository/User.php');
4
5   use Repository\User as UserRepository;
6
7   class EntityManager
8   {
9           private $host;
10          private $db;
11          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);
33          }
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  }
```

Der Entity Manager wird damit zum zentralen Einstiegspunkt, sowohl für die Erzeugung der Datenbankverbindung, als auch für die späteren Abfragen. Nach allen Umbauten ist das angezeigte Ergebnis nach wie vor identisch:

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

Jetzt haben wir schon eine ganze Menge Code geschrieben - und das nur, um die Daten aus der Datenbanktabelle in ein Objekt zu überführen. An der Anwendung haben wir hingegen so gut wie noch gar nicht gearbeitet. Es zeigt sich bereits: Die Implementierung eines ORM ist ein "dickes Brett", die Realisierung wird aufwändig. Und wir haben ja gerade erst einmal angefangen. Aber machen wir noch einmal ein bisschen weiter, damit wir Doctrine 2 später auch wirklich richtig zu schätzen wissen.

3.2 Speichern einer Entity

Bislang haben wir uns einen sehr trivialen Anwendungsfall angesehen: Wir holen ein einziges Objekt auf Basis seiner ID aus einer einzigen Datenbanktabelle. Wie gehen wir aber mit Schreiboperationen um? Grundsätzlich gibt es zwei Fälle: Entweder, es wird ein neues Objekt erstellt, oder aber, ein existierendes Objekt wird modifiziert. Je nach dem muss dann entweder ein Insert- oder aber ein Update-Statement auf der Datenbank ausgeführt werden. Implementieren wir zunächst den "Insert-Fall". Dazu erweitern wir zunächst die den User-Mapper um eine Methode `extract()`:

```php
<?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)
                                );
                        }
                }

```

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

Auf diesem Wege bekommen wir die zu speichernden Daten also aus dem entsprechenden Objekt heraus. Wenn wir nun noch den Entity Manager um eine `saveUser()`-Methode erweitern, können wir neue Datensätze einfügen:

```
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  }
```

Das Einfügen eines neuen Datensatzes funktioniert nun wie folgt:

```
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('Musermann');
10 $newUser->setGender(1);
11 $em->saveUser($newUser);
12
13 echo $newUser->assembleDisplayName();
```

So weit, so gut! Was nun aber, wenn wir ein existierendes Objekt / einen existierenden Datensatz manipulieren wollen? Woher wissen wir, ob wir es mit einem neuen oder einem geänderten Datensatz zu tun haben? In unserem Fall wäre es sicherlich legitim, wenn wir prüfen würden, ob ein Wert für "id" gesetzt ist, denn schließlich handelt es sich ja um ein Feld mit "auto_increment" - die Datenbank vergibt hier also selbst den notwendigen Wert beim initialen Speichern. So richtig wasserdicht oder elegant ist diese Lösung allerdings nicht. Außerdem könnten wir unter Verwendung einer sog. *Identity Map* noch weitere Vorteile erzielen: So lässt sich etwa das wiederholte Laden zuvor bereits geladener Entities vermeiden. Die Identity Map ist dabei nichts anderes als ein assoziatives Array, dass Referenzen auf bereits geladene Objekte hält. Ein guter Ort dafür ist der Entity Manager, den ich auch gleich um die entsprechende "Lookup-Logik" in der `saveUser()`-Methode erweitere:

```php
<?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)
```

```
33          {
34                  return $this->connection->query($stmt);
35          }
36
37          public function saveUser($user)
38          {
39                  $userMapper = new UserMapper();
40                  $data = $userMapper->extract($user);
41
42                  $userId = call_user_func(
43                          array($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  }
```

Den User-Mapper habe ich zudem noch um die Methode `getIdColumn()` erweitert, die den Wert "id" zurückliefert. Nun kann ich im Programm selbst Folgendes ausführen:

```
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);
```

Der Datensatz wird korrekt in der Datenbank geändert, es wird kein weiterer Datensatz hinzugefügt.

3.3 Assoziationen

Nun wollen wir die Übersicht aller Posts eines Nutzers generieren. Ich stelle mir vor, dass wir später so auf einfache Weise über die Beiträge eines Nutzers iterieren und jeweils die Überschrift des Beitrags anzeigen können:

```php
<?php
include_once('../EntityManager.php');
$em = new EntityManager('localhost', 'app', 'root', '');
$user = $em->getUserRepository()->findOneById(1);
?>
<h1><?echo $user->assembleDisplayName(); ?></h1>
<ul>
<?php foreach($user->getPosts() as $post) { ?>
<li><?php echo $post->getTitle(); ?></li>
<?php } ?>
</ul>
```

Was müssen wir dafür alles tun? Zunächst einmal erstellen wir eine passende Datenstruktur:

```sql
CREATE TABLE posts(
        id int(10) NOT NULL auto_increment,
        user_id int(10) NOT NULL,
        title varchar(255) NOT NULL,
        content text NOT NULL,
        PRIMARY KEY (id)
);
```

Zudem legen wir dort für den Testnutzer auch Testposts ab, so dass wir unsere Implementierung später auch testen können. Jetzt gilt es eine ganze Reihe von Klassen zu erweitern. Die User-Entity erweitern wir um die `getPosts()`-Methode, die beim ersten Aufruf für das Nachladen der Beiträge des entsprechenden Nutzers über das entsprechende Repository sorgt:

```php
<?php
namespace Entity;

class User
{
        // [..]

    private $postRepository;

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

```

```
16              return $this->posts;
17          }
18
19          // [..]
20  }
```

Damit das funktioniert, muss der User-Entity allerdings das `PostRepository` zur Verfügung stehen. Dazu muss das User-Repository entsprechend in der Methode `findOneById()` erweitert werden:

```
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  }
```

Und dazu muss wiederum der `EntityManager` den Zugriff bieten:

```php
<?php

// [..]

class EntityManager
{
        // [..]

        private $postRepository;

        public function getPostRepository()
        {
                if (!is_null($this->postRepository)) {
                        return $this->postRepository;
                } else {
                        $this->postRepository = new PostRepository($this);
                        return $this->postRepository;
                }
        }
}
```

Nun müssen die notwendigen Klassen, die jetzt bereits oben verfügbar gemacht und verwendet werden, aber natürlich selbst noch entwickelt werden. Zunächst die Post-Entity selbst:

```php
<?php
namespace Entity;

class Post
{
        private $id;
        private $title;
        private $content;

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

```
14
15          public function getContent()
16          {
17                  return $this->content;
18          }
19
20          public function setId($id)
21          {
22                  $this->id = $id;
23          }
24
25          public function getId()
26          {
27                  return $this->id;
28          }
29
30          public function setTitle($title)
31          {
32                  $this->title = $title;
33          }
34
35          public function getTitle()
36          {
37                  return $this->title;
38          }
39  }
```

Dann der obligatorische Mapper:

```
1   <?php
2   namespace Mapper;
3
4   class Post
5   {
6           private $mapping = array(
7                   'id' => 'id',
8                   'title' => 'title',
9                   'content' => 'content',
10          );
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  }
```

Und nicht zu vergessen, das Post-Repository:

```php
<?php
namespace Repository;

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

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;
        }
}
```

Alles in allem stellt sich das Anwendung-Verzeichnis im Dateisystem bis zu diesem Punkt also wie folgt dar:

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

Code-Download

Den Code bis zu diesem Zeitpunkt findest du bei github zum Download[1].

3.4 Ausblick & Fazit

Prima, das haben wir soweit hinbekommen. So richtig gut sieht das alles aber bereits jetzt schon nicht mehr aus. Ein Problem ist die Tatsache, dass sich technischer und fachlicher Code bereits sehr stark mischt. Viele der Lösungen sind zudem spezifisch nur für das jeweilige Problem, einige Codestellen riechen nach “Copy & Paste” und müssten bereits jetzt wieder vollständig refactored werden.

Was ist jetzt aber mit zusammengesetzten Primärschlüsseln, dem Löschen und Hinzufügen von Assoziationen, n:m-Beziehungen, Vererbung, Performance-Optimierungen, Caching, den Entities, Mappern und Repositories für die fehlenden zig weiteren fachlichen Objekte der Anwendung und so weiter? Und was passiert, wenn sich die Datenstrukturen der Anwendung auch nur ein kleinwenig ändern müssen? Dann müssten wir im Grunde zig Klassen ändern. Es ist wohl endgültig an der Zeit, dass wir uns Doctrine 2 ansehen.

[1] https://github.com/michael-romer/doctrine2buch

4 Hallo, Doctrine 2! (verfügbar)

Keine Sorge ...

Im Laufe dieses Buches gehen wir noch alle Themen, die wir im Folgenden teils bereits kurz anreißen, noch einmal im Detail ein.

4.1 Installation

Am einfachsten lässt sich Doctrine 2 installieren, wenn man das Tool "Composer" zur Hilfe nimmt. Dazu lädt man einmal Composer von dessen Website[1] herunter und legt die Phar-Datei im Root-Verzeichnis der eigenen Anwendung ab. Zusätzlich erstellt man die Datei `composer.json` mit folgendem Inhalt:

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

Das Kommando

```
1 $ php composer.phar install
```

im Root-Verzeichnis der Anwendung sorgt nun dafür, dass Doctrine 2 heruntergeladen und verfügbar gemacht wird. Dazu erzeugt Composer ein neues Verzeichnis `vendor`. Aber Composer lädt nicht nur den Code herunter, sondern richtet gleichzeitig auch das Autoloading der Doctrine-Klassen ein, so dass an geeigneter Stelle, etwa früh in der `index.php`, nur noch die folgenden Zeilen hinzugefügt werden müssen:

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

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

4.2 Die erste Entity

Ausgehend von ersten Kapitel dieses Buch, in dem wir ein eigenes ORM-System zu programmieren angefangen haben, können wir nun vieles radikal vereinfachen, wenn wir die User-Entity wie folgt mit Annotationen aus der Doctrine 2 Bibliothek versehen:

```php
<?php
namespace Entity;

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

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

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

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

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

        const GENDER_MALE = 0;
        const GENDER_FEMALE = 1;
```

```
30
31          const GENDER_MALE_DISPLAY_VALUE = "Mr.";
32          const GENDER_FEMALE_DISPLAY_VALUE = "Mrs.";
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 }
```

Die `index.php` kann wie folgt geändert werden, um die Entity via Doctrine zu lesen und zu ändern:

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

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

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

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

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

4.3 Die erste Assoziation

Und das war es schon. Auch die Assoziation ist kein größeres Problem. Dazu muss die User-Entity noch einmal wie folgt erweitert werden:

```php
<?php
namespace Entity;

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

```

```
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  }
```

Und die notwendigen Annotationen bei der Post-Entity:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="posts")
7   */
8   class Post
9   {
10          /**
11          * @Id @Column(type="integer")
12          * @GeneratedValue
13          */
14          private $id;
15
16          /**
17          * @ManyToOne(targetEntity="Entity\User", inversedBy="posts")
18          * @JoinColumn(name="user_id", referencedColumnName="id")
19          */
20          private $user;
21
22          /** @Column(type="string") */
23          private $title;
24
25          /** @Column(type="string") */
```

```
26          private $content;
27
28          public function setUserId($user_id)
29          {
30                  $this->user_id = $user_id;
31          }
32
33          public function getUserId()
34          {
35                  return $this->user_id;
36          }
37
38          public function setContent($content)
39          {
40                  $this->content = $content;
41          }
42
43          public function getContent()
44          {
45                  return $this->content;
46          }
47
48          public function setId($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  }
```

Sobald der entsprechende User geladen ist, kann wie gehabt über die Posts des Nutzers iteriert werden:

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

Alles in allem kommen wir nun allerdings mit deutlich weniger Klassen aus:

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

Einfacher geht es kaum - Doctrine 2 sei Dank! Bevor wir Talking im Folgenden Stück für Stück weiterentwickeln, schauen wir uns zunächst ein paar theoretische Konzepte von Doctrine 2 an.

Code-Download

Den Code bis zu diesem Zeitpunkt findest du bei github zum Download[2].

[2] https://github.com/michael-romer/doctrine2buch/tree/doctrine