DOCTRINE

na prática

Elton Luís Minetto

Doctrine na prática

eminetto

Esse livro está à venda em http://leanpub.com/doctrine-na-pratica

Essa versão foi publicada em 2014-09-05

Conteúdo

Introdução

Projeto Doctrine

O Doctrine é um projeto *Open Source* que tem por objetivo criar uma série de bibliotecas *PHP* para tratar das funcionalidades de persistência de dados e funções relacionadas a isto.

O projeto é dividido em alguns sub-projetos, sendo os dois mais importantes o *Database Abstraction Layer* (*DBAL*) e o *Object Relational Mapper* (*ORM*).

Database Abstraction Layer

Construido sob o *PHP Data Objects* (*PDO*) o *DBAL* provê uma camada de abstração que facilita a manipulação dos dados usando uma interface orientada a objetos. Por usar o *PDO* como suporte é necessário termos as extensões configuradas. Se formos usar o *DBAL* para acessar uma base de dados *MySQL*, por exemplo, é necessário instalar a extensão correspondente.

Além da manipulação de dados (*insert*, *update*, etc) o pacote *DBAL* nos fornece outras funcionalidades importantes como introspecção da base de dados (podemos obter informações da estrutura de tabelas e campos), transações, eventos, etc. Vamos ver algumas destas funcionalidades nos próximos capítulos.

Object Relational Mapper

Vou usar aqui a definição que encontra-se na *Wikipedia* pois ela resume bem o conceito do *ORM*:

> [...]é uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada aos objetos utilizando bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes. Com esta técnica, o programador não precisa se preocupar com os comandos em linguagem SQL; ele irá usar uma interface de programação simples que faz todo o trabalho de persistência.

Os *ORMs* são usados em diversas linguagens de programação e ambientes para facilitar o uso de banco de dados e manter uma camada de abstração entre diferentes bases de dados e conceitos.

O *Doctrine* tornou-se o "padrão de facto" para solucionar o problema do mapeamento objeto relacional no ambiente PHP e vem sendo usado por projetos de diversos tamanhos e frameworks como o Symfony. No decorrer dos capítulos deste *e-book* vamos aprender a usá-lo para este fim.

Instalação

A forma mais fácil de instalarmos o *Doctrine* é usando o *Composer*. O *Composer* é um gerenciador de dependências para PHP. Com ele podemos definir quais pacotes vamos usar no nosso projeto e gerenciar a instalação e atualização dos mesmos.

O primeiro passo é instalarmos o próprio *Composer*. No *Linux* ou *MacOSX* é possível instalar o *Composer* pela linha de comando, executando o comando, dentro do diretório do nosso projeto:

```
curl -sS https://getcomposer.org/installer | php
```

Outra opção é executar o comando abaixo, que não depende do pacote *curl*:

```
php -r "eval('?>'.file_get_contents('https://getcomposer.org/installer'));"
```

No *Windows* é possível fazer o download do *Composer* pela url http://getcomposer.org/composer.phar[1] ou usar o instalador binário, conforme mostra a documentação oficial[2].

Com o *Composer* instalado precisamos configurá-lo para detalhar quais pacotes vamos usar. Para isso basta criarmos um arquivo chamado *composer.json* na raiz do projeto. No nosso caso vamos definir o uso dos pacotes do projeto *Doctrine* e suas versões mais atuais no momento da escrita deste livro, a 2.4.X.:

```
{
    "require": {
        "doctrine/common": "2.4.*",
        "doctrine/dbal": "2.4.*",
        "doctrine/orm": "2.4.*"
    }
}
```

Você pode encontrar outros pacotes disponíveis para o *Composer* fazendo uma pesquisa no diretório oficial de pacotes, no site https://packagist.org[3].

Com o arquivo *composer.json* criado podemos executar o comando para que a instalação seja feita:

[1] http://getcomposer.org/composer.phar
[2] http://getcomposer.org/doc/00-intro.md#installation-windows
[3] https://packagist.org

```
php composer.phar install
```

Criando o bootstrap.php

Vamos agora criar o *bootstrap* do nosso projeto. Este arquivo possui este nome pois é usado para inicializar e configurar o ambiente, dar o "pontapé inicial" do projeto. Ele vai ser executado todas as vezes que executarmos algum script, por isso é importante especial atenção a este arquivo, para que ele não contenha erros ou processamentos pesados que possam deixar as coisas lentas.

```php
<?php
//AutoLoader do Composer
$loader = require __DIR__.'/vendor/autoload.php';
//vamos adicionar nossas classes ao AutoLoader
$loader->add('DoctrineNaPratica', __DIR__.'/src');


use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;
use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\Common\Annotations\AnnotationRegistry;

//se for falso usa o APC como cache, se for true usa cache em arrays
$isDevMode = false;

//caminho das entidades
$paths = array(__DIR__ . '/src/DoctrineNaPratica/Model');
// configurações do banco de dados
$dbParams = array(
    'driver'   => 'pdo_mysql',
    'user'     => 'root',
    'password' => '',
    'dbname'   => 'dnp',
);

$config = Setup::createConfiguration($isDevMode);

//leitor das annotations das entidades
$driver = new AnnotationDriver(new AnnotationReader(), $paths);
$config->setMetadataDriverImpl($driver);
//registra as annotations do Doctrine
```

```
33 AnnotationRegistry::registerFile(
34     __DIR__ . '/vendor/doctrine/orm/lib/Doctrine/ORM/Mapping/Driver/DoctrineAnno\
35 tations.php'
36 );
37 //cria o entityManager
38 $entityManager = EntityManager::create($dbParams, $config);
```

https://gist.github.com/eminetto/7312206[4]

Tentei documentar as principais funções do arquivo nos comentários do código, mas vou destacar alguns pontos importantes.

- As primeiras duas linhas de código são importantes pois carregam o *autoloader* do *Composer* e o configura para reconhecer as classes do projeto, que iremos criar no decorrer do livro. O *autoloader* é responsável por incluir os arquivos PHP necessários sempre que fizermos uso das classes definidas na sessão *use* dos arquivos.
- Na linha 18 definimos onde vão ficar as classes das nossas *entidades*. Neste contexto, entidades são a representação das tabelas da nossa base de dados, que serão usadas pelo *ORM* do Doctrine. Iremos criar estas classes no próximo capítulo.
- O código entre a linha 30 e a 36 é responsável por configurar as *Annotations* do Doctrine. Como veremos no próximo capítulo existe mais de uma forma de configurarmos as entidades (*YAML* e *XML*) mas vamos usar neste livro o formato de anotações de blocos de códigos, que é uma das formas mais usadas.
- A linha 38 cria uma instância do *EntityManager*, que é o componente principal do *ORM* e como seu nome sugere é o responsável pela manipulação das entidades (criação, remoção, atualização, etc). Iremos usá-lo inúmeras vezes no decorrer deste livro.

Precisamos agora criar a estrutura de diretórios onde iremos salvar as classes das nossas entidades, conforme configurado na linha 18 do *bootstrap.php*. Esta estrutura de diretórios segue o padrão *PSR*[5] que é usado pelos principais frameworks e projetos, inclusive o próprio Doctrine. No comando abaixo, do Linux/MacOSX, criamos o diretório *src* dentro da raiz do nosso projeto:

```
1 mkdir -p src/DoctrineNaPratica/Model
```

Na linha 20 do *bootstrap.php* configuramos o Doctrine para conectar em uma base de dados *MySQL* chamada *dnp*. O Doctrine consegue criar as tabelas representadas pelas entidades, mas não consegue criar a base de dados, pois isso é algo que depende bastante do sistema gerenciador de base de dados. Por isso vamos criar a base de dados para nosso projeto, no *MySQL*:

[4] https://gist.github.com/eminetto/7312206
[5] https://github.com/php-fig/fig-standards/tree/master/accepted

```
mysql -uroot
create database dnp;
```

Caso esteja usando outro sistema gerenciador de banco de dados como o *PostgreSQL*, *Oracle*, *SQLite*, etc, é necessário que você verifique o a necessidade ou não de criar uma base de dados antes de passar para os próximos passos.

Configurar a ferramenta de linha de comando

Um dos recursos muito úteis do Doctrine é a sua ferramenta de linha de comando, que fornece funcionalidades de gerenciamento como criar tabelas, limpar cache, etc. O primeiro passo é criarmos o arquivo de configuração da ferramenta, o *cli-config.php*, na raiz do nosso projeto:

```
<?php
// cli-config.php
require_once 'bootstrap.php';

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

https://gist.github.com/eminetto/7312213[6]

Como podemos ver, ele faz uso do *bootstrap.php* e cria uma instância da classe *HelperSet* que é usada pelo próprio Doctrine, na ferramenta de linha de comandos.

Podemos testar se configuramos tudo da maneira correta executando:

```
./vendor/bin/doctrine
```

Ou, no caso do windows:

```
php vendor/bin/doctrine.php
```

Se tudo estiver correto você verá uma lista de comandos disponíveis e uma pequena ajuda explicando como usá-los. Iremos usar alguns deles nos próximos capítulos.

[6] https://gist.github.com/eminetto/7312213

Definindo o projeto

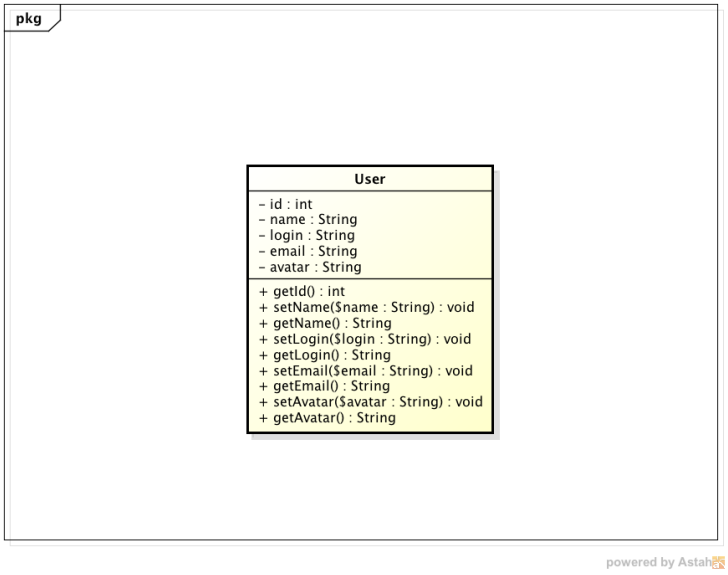
Neste capítulo vamos começar a ver alguns conceitos do Doctrine, como o relacionamento entre entidades e também como manipulá-las.

Vamos primeiro definir um caso para servir como estudo. Criaremos um sistema de controle de alunos com seus perfis de redes sociais, cursos, lições e o progresso de aprendizado. Nas próximas páginas vamos ver alguns diagramas e códigos para entendermos o projeto e como desenvolvê-lo usando o Doctrine.

Criando as entidades

User

Começaremos pela entidade *User*, que representa tanto nossos alunos como os professores. Primeiro vamos criar uma versão bem simples, sem relacionamentos:



User

Com a entidade modelada podemos definí-la em *PHP*, usando uma linguagem que o Doctrine a entenda. O arquivo *src/DoctrineNaPratica/Model/User.php* ficou desta forma:

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="User")
*/
class User
{
        /**
         * @ORM\Id @ORM\Column(type="integer")
         * @ORM\GeneratedValue
         * @var integer
         */
        protected $id;

        /**
         * @ORM\Column(type="string", length=150)
         *
         * @var string
         */
        private $name;

        /**
         * @ORM\Column(type="string", length=20, unique=true)
         *
         * @var string
         */
        private $login;

        /**
         * @ORM\Column(type="string", length=150, nullable=true)
         *
         * @var string
         */
        private $email;

        /**
         * @ORM\Column(type="string", length=255, nullable=true)
         *
```

```
43          * @var string
44          */
45         private $avatar;
46
47         /**
48          * @return integer
49          */
50         public function getId()
51         {
52             return $this->id;
53         }
54
55         public function getName()
56         {
57             return $this->name;
58         }
59
60         public function setName($name)
61         {
62             return $this->name = $name;
63         }
64
65         public function getLogin()
66         {
67             return $this->login;
68         }
69
70         public function setLogin($login)
71         {
72             return $this->login = $login;
73         }
74
75         public function getEmail()
76         {
77             return $this->email;
78         }
79
80         public function setEmail($email)
81         {
82             return $this->email = $email;
83         }
84
```

```
85          public function getAvatar()
86          {
87              return $this->avatar;
88          }
89
90          public function setAvatar($avatar)
91          {
92              return $this->avatar = $avatar;
93          }
94  }
```

https://gist.github.com/eminetto/7312217[7]

A primeira decisão que temos que tomar é a forma como iremos definir as classes para que o Doctrine as entenda como entidades. É possível fazermos isto de quatro formas:

- usando anotações nos blocos de código (Docblock Annotations)
- usando arquivos *XML*
- usando arquivos *YAML*
- usando *arrays* em PHP (o menos usado)

No arquivo acima é possível ver que usei as anotações em blocos de código. Desta forma, o comentário *@ORMEntity* define que a classe é uma entidade, e as outras tags definem coisas como o nome da tabela (*@ORMTable(name="User")*) ou os tipos dos campos (*@ORMColumn(type="string", length=150)*), etc. No capítulo anterior nós configuramos o ambiente para usar as anotações, conforme é possível ver na linha 30 do *bootstrap.php*. Caso deseje usar outra forma de definição, como *XML*, é preciso alterar a classe *User* e também o *bootstrap.php* para refletir esta mudança. Eu prefiro usar as anotações porque assim a definição da classe e seu comportamento no Doctrine estão contidas em um mesmo arquivo, o que facilita a manutenção do projeto. Caso você opte por usar outra forma de configuração é preciso alterar o arquivo PHP e um arquivo *.yml* ou *.xml* para cada mudança na modelagem, o que eu acho pouco produtivo.

Outro ponto que é importante entendermos é a definição dos tipos de dados da entidade. Precisamos configurar cada um dos atributos da nossa classe para um tipo de dados que o Doctrine consiga traduzir para os mais diversos bancos de dados suportados. Os principais tipos suportados no momento da escrita deste livro são:

[7] https://gist.github.com/eminetto/7312217

Annotation	PHP	SQL
string	string	VARCHAR
integer	int	INT
smallint	int	SMALLINT
bigint	string	BIGINT
boolean	boolean	BOOLEAN
decimal	string	DECIMAL
date	DateTime	DATETIME
time	DateTime	TIME
datetime	DateTime	DATETIME/TIMESTAMP
text	string	CLOB
object	Object	CLOB
float	double	FLOAT
blob	Resource Stream	BLOB

Mapping

Além destes tipos básicos é possível criar novos, extendendo classes do Doctrine.

Outra configuração importante da classe *User* é a tag *@ORMGeneratedValue* no atributo *id* que define que este é um campo gerado automaticamente, o que o Doctrine vai mapear para um AUTO_-INCREMENT no *MySQL* ou o correspondente em outros bancos de dados.

Além das anotações de configuração do Doctrine, na classe *User* também criamos os *getters* e *setters*, métodos que são responsáveis por permitir o acesso aos nossos atributos, como o *getName()* e o *setName()*. Vamos criar métodos parecidos para todos os atributos das classes, pois isso é uma boa prática quando estamos trabalhando com objetos.

Vamos agora usar a ferramenta de linha de comando do Doctrine, que configuramos no capítulo anterior. Com ela podemos mandar o Doctrine ler a nossa definição de classe e criar a tabela correspondente na base de dados. Para isso basta executar o comando:

```
1 ./vendor/bin/doctrine orm:schema-tool:create
```

O Doctrine vai verificar sua configuração de banco de dados e a sua definição de entidade. Se tudo estiver correto a tabela vai ser criada.

Outra funcionalidade útil da linha de comando é podermos também validar se nosso banco de dados está sincronizado com as definições de entidade, bem como verificar se a configuração dos relacionamentos está correta (vamos ver sobre os relacionamentos nas próximas páginas).

```
1 ./vendor/bin/doctrine orm:validate-schema
```

Caso algo esteja errado você ira receber uma lista de erros a corrigir.

E também ver as informações sobre as entidades criadas:

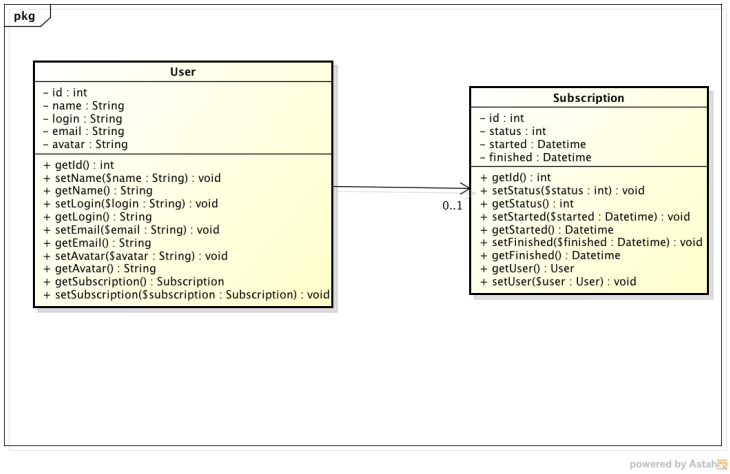
```
./vendor/bin/doctrine orm:info
```

Relacionamentos

Vamos agora ver como o Doctrine trata o relacionamento entre as entidades. Vamos incrementando o nosso projeto passo a passo, e adicionando novas entidades e aprendendo como elas se relacionam.

Subscription

Vamos criar a entidade *Subscription*, conforme a modelagem a seguir. Um usuário pode se matricular apenas uma vez na nossa "universidade" e uma matrícula (*subscription*) pertence a apenas um usuário, o que figura um relacionamento *OneToOne* (um para um).



UserSubscription

Para que o Doctrine entenda a relação entre estas duas entidades precisamos configurar em ambas as classes. Vamos começar alterando a classe *User*, adicionando o atributo *$subscription*, seu *getter*, *setter* e a anotação que indica a sua relação com a entidade *Subscription*:

```
/**
 * @ORM\OneToOne(targetEntity="Subscription", mappedBy="user", cascade={"all"})
*/
private $subscription;

public function getSubscription()
{
        return $this->subscription;
}
public function setSubscription($subscription)
```

```
11  {
12          return $this->subscription = $subscription;
13  }
```

https://gist.github.com/eminetto/7312232[8]

A novidade no código acima é justamente a anotação *@ORMOneToOne*. Ela possui quatro parâmetros:

- targetEntity: indica qual é a entidade a que este campo faz referência, neste caso a *Subscription*.
- mappedBy: indica qual é o campo que irá existir na classe *Subscription* que faz referência a esta classe, neste caso vai ser o campo *user*.
- cascade: diz ao Doctrine que comportamento ele deve tomar quando um *User* for removido da base de dados. Neste caso ele vai propagar essa mudança para a entidade *Subscription*, eliminando-a também.

Vamos agora criar a entidade *Subscription* no arquivo *src/DoctrineNaPratica/Model/Subscription.php*:

```
1   <?php
2   namespace DoctrineNaPratica\Model;
3
4   use Doctrine\ORM\Mapping as ORM;
5
6   /**
7    * @ORM\Entity
8    * @ORM\Table(name="Subscription")
9    */
10  class Subscription
11  {
12
13          /**
14           * @ORM\Id @ORM\Column(type="integer")
15           * @ORM\GeneratedValue
16           * @var integer
17           */
18          private $id;
19
20
21          /**
22           * @ORM\OneToOne(targetEntity="User", inversedBy="subscription")
```

[8] https://gist.github.com/eminetto/7312232

```
23          * @ORM\JoinColumn(name="user_id", referencedColumnName="id")
24          **/
25         private $user;
26
27
28         /**
29          * @ORM\Column(type="integer", nullable=false)
30          *
31          * @var int
32          */
33         private $status;
34
35         /**
36          * @ORM\Column(type="datetime")
37          * @var datetime
38          */
39         protected $started;
40
41         /**
42          * @ORM\Column(type="datetime",nullable=true)
43          * @var datetime
44          */
45         protected $finished;
46
47         /**
48          * @return integer
49          */
50         public function getId()
51         {
52             return $this->id;
53         }
54
55         public function getUser()
56         {
57             return $this->user;
58         }
59
60         public function setUser($user)
61         {
62             $this->user = $user;
63         }
64
```

```
65          public function getStatus()
66          {
67              return $this->status;
68          }
69
70          public function setStatus($status)
71          {
72              return $this->status = $status;
73          }
74
75          public function getStarted()
76          {
77              return $this->started;
78          }
79
80          public function setStarted($started)
81          {
82              return $this->started = $started;
83          }
84
85          public function getFinished()
86          {
87              return $this->finished;
88          }
89
90          public function setFinished($finished)
91          {
92              return $this->finished = $finished;
93          }
94  }
```

https://gist.github.com/eminetto/7312237[9]

Também fazemos a definição do relacionamento entre as duas entidades, usando a anotação *OneToOne* com os parâmetros:

- targetEntity: indica qual é a entidade relacionada, no caso a *User*
- inversedBy: indica qual é o campo da entidade *User* que faz referência a esta *Subscription*, neste caso o atributo *subscription*.

Logo abaixo usamos outra anotação, a *JoinColumn*, que é opcional, mas está indicando que será criada na tabela *Subscription* um campo chamado user_id que faz a chave estrangeira com a tabela

[9] https://gist.github.com/eminetto/7312237

User. Esta anotação é opcional pois o comportamento padrão é criar um campo com o nome da tabela seguido por _id. Caso você queira mudar este comportamento pode usar a anotação *JoinColumn* para alterar o nome do campo.

Vamos agora atualizar a base de dados com a nova tabela e as suas relações executando:

```
1 ./vendor/bin/doctrine orm:schema-tool:update --force
```

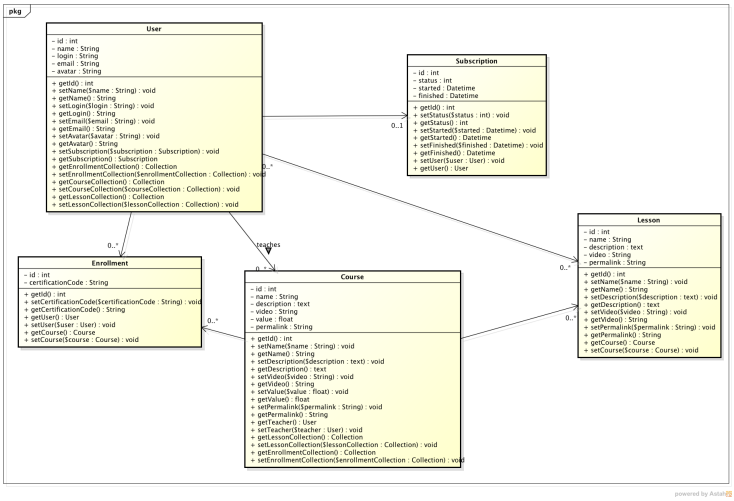
O parâmetro *–force* é necessário pois a base de dados já existe. O Doctrine vai avisá-lo que essa é uma operação potencialmente perigosa, mas sabemos o que estamos fazendo neste caso, pois nossa aplicação ainda está em fase de desenvolvimento.

É possível também apenas visualizar o comando *SQL* responsável pela alteração:

```
1 ./vendor/bin/doctrine orm:schema-tool:update --dump-sql
```

Cursos e lições

Vamos acrescentar mais funcionalidades ao nosso projeto. A próxima imagem mostra a nova modelagem.



UserSubscriptionCourseLesson

Um usuário pode se matricular em mais de um curso e um curso possui vários alunos, o que figura um relacionamento *ManyToMany* (muitos para muitos). O mesmo ocorre entre as entidades *Lesson* e *User* (um usuário pode cursar várias lições e uma lição possui vários alunos). Para configurar a relação entre *Lesson* e *User* vamos usar a anotação *@ORMManyToMany*, que irá criar automaticamente uma tabela para servir como ligação, a *LessonUser* que definimos na linha 69 da nova versão da classe *User*, mostrada a seguir.

Quanto a relação *ManyToMany* entre *Course* e *User* vamos precisar usar outra abordagem. Como precisamos que a tabela de ligação entre as duas entidades possua campos extras (*certificationCode* neste caso) precisamos criar a entidade de ligação manualmente, e usar as anotações *OneToMany* e *ManyToOne*. O Doctrine não permite que uma tabela de ligação gerada automaticamente (como a *LessonUser*) possua campos extras além dos responsáveis pela chave estrangeira. Esta é a função da entidade *Enrollment* no nosso projeto.

Vamos agora observar o código atualizado da classe *User*, com todas as relações mostradas na imagem anterior.

```
1  <?php
2  namespace DoctrineNaPratica\Model;
3
4  use Doctrine\ORM\Mapping as ORM;
5
6  /**
7   * @ORM\Entity
8   * @ORM\Table(name="User")
9   */
10 class User
11 {
12         /**
13          * @ORM\Id @ORM\Column(type="integer")
14          * @ORM\GeneratedValue
15          * @var integer
16          */
17         protected $id;
18
19         /**
20          * @ORM\Column(type="string", length=150)
21          *
22          * @var string
23          */
24         private $name;
25
26         /**
27          * @ORM\Column(type="string", length=20, unique=true)
28          *
29          * @var string
30          */
31         private $login;
32
33         /**
```

```
34          * @ORM\Column(type="string", length=150, nullable=true)
35          *
36          * @var string
37          */
38         private $email;
39
40         /**
41          * @ORM\Column(type="string", length=255, nullable=true)
42          *
43          * @var string
44          */
45         private $avatar;
46
47         /**
48          * @ORM\OneToOne(targetEntity="Subscription", mappedBy="user")
49          **/
50         private $subscription;
51
52         /**
53          * @ORM\OneToMany(targetEntity="Enrollment", mappedBy="user", cascade={"all"}, \
54 orphanRemoval=true, fetch="LAZY")
55          */
56         private $enrollmentCollection;
57
58         /**
59          * @ORM\OneToMany(targetEntity="Course", mappedBy="teacher", cascade={"all"}, o\
60 rphanRemoval=true, fetch="LAZY")
61          *
62          * @var Doctrine\Common\Collections\Collection
63          */
64         protected $courseCollection;
65
66         /**
67          * @ORM\ManyToMany(targetEntity="Lesson", inversedBy="userLessons", cascade={"a\
68 ll"})
69          * @ORM\JoinTable(name="LessonUser",
70          *   joinColumns={@ORM\JoinColumn(name="user_id", referencedColumnName="id")},
71          *   inverseJoinColumns={@ORM\JoinColumn(name="lesson_id", referencedColumnName\
72 ="id")}
73          * )
74          */
75         private $lessonCollection;
```

```
76
77
78          /**
79           * @return integer
80           */
81          public function getId()
82          {
83              return $this->id;
84          }
85
86          public function getName()
87          {
88              return $this->name;
89          }
90
91          public function setName($name)
92          {
93              return $this->name = $name;
94          }
95
96          public function getLogin()
97          {
98              return $this->login;
99          }
100
101         public function setLogin($login)
102         {
103             return $this->login = $login;
104         }
105
106         public function getEmail()
107         {
108             return $this->email;
109         }
110
111         public function setEmail($email)
112         {
113             return $this->email = $email;
114         }
115
116         public function getAvatar()
117         {
```

```
118             return $this->avatar;
119         }
120 
121         public function setAvatar($avatar)
122         {
123             return $this->avatar = $avatar;
124         }
125 
126         public function getSubscription()
127         {
128             return $this->subscription;
129         }
130 
131         public function setSubscription($subscription)
132         {
133             return $this->subscription = $subscription;
134         }
135 
136         public function getEnrollmentCollection()
137         {
138             return $this->enrollmentCollection;
139         }
140 
141         public function setEnrollmentCollection($enrollmentCollection)
142         {
143             return $this->enrollmentCollection = $enrollmentCollection;
144         }
145 
146         public function getCourseCollection()
147         {
148             return $this->courseCollection;
149         }
150 
151         public function setCourseCollection($courseCollection)
152         {
153             return $this->courseCollection = $courseCollection;
154         }
155 
156         public function getLessonCollection()
157         {
158             return $this->lessonCollection;
159         }
```

```
160
161         public function setLessonCollection($lessonCollection)
162         {
163             return $this->lessonCollection = $lessonCollection;
164         }
165
166         public function __construct()
167         {
168             $this->courseCollection = new ArrayCollection;
169             $this->lessonCollection = new ArrayCollection;
170             $this->enrollmentCollection = new ArrayCollection;
171         }
172 }
```

https://gist.github.com/eminetto/7312242[10]

Na Linha 53 estamos usando a configuração *fetch="LAZY"*. Isto diz ao Doctrine como deverá fazer a busca dos dados da entidade relacionada (a *Enrollment*). As opções são *EAGER* (os dados da entidade relacionada vão ser recuperadas sempre), *LAZY* (os dados da entidade relacionada só serão carregados do banco de dados quando o método *getEnrollmentCollection* for chamado) e *EXTRA_LAZY* (permite que os dados sejam carregados parcialmente usando alguns métodos da classe *ArrayCollection* do Doctrine).

As configurações de relacionamento com a entidade *Enrollment* e a *Course* são parecidas com as que fizemos no tópico anterior. A diferença é que agora usamos o relacionamento *OneToMany*.

Já com a *Lesson* estamos usando a *ManyToMany*, que é um pouco mais complexa. Além da definição dos nomes das entidades e os campos de cada uma que fazem referência ao outro lado da relação estamos também definindo os detalhes da tabela de ligação que será criada, a *LessonUser*. A anotação é bem explicativa mas, resumindo, estamos definindo os nomes dos campos que são criados para referencias as tabelas de destino.

Outro ponto importante na nova versão da classe *User* é a criação de um construtor (o método *__construct*) que é responsável pela inicialização das coleções de dados que serão usadas pelo Doctrine (*lessonCollection*, *courseCollection* e *enrollmentCollection*). Elas são instâncias da classe *ArrayCollection* do Doctrine, que possui vários métodos que auxiliam o tratamento de coleções de objetos, como busca e ordenação, entre outros.

Course

A seguir vemos o código da classe *Course*, com a descrição de seus atributos e relações entre as outras entidades (*User*, *Lesson* e *Enrollment*). A classe deve ser criada no arquivo *src/DoctrineNaPratica/Model/Course.php*.

[10] https://gist.github.com/eminetto/7312242

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="Course")
 */
class Course
{
        /**
         * @ORM\Id @ORM\Column(type="integer")
         * @ORM\GeneratedValue
         * @var integer
         */
        protected $id;

        /**
         * @ORM\Column(type="string", length=150)
         *
         * @var string
         */
        private $name;

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

        /**
         * @ORM\Column(type="text", nullable=true)
         *
         * @var string
         */
        private $video;

        /**
         * @ORM\Column(type="string", length=255, nullable=true)
         *
```

```
43          * @var string
44          */
45         private $permalink;
46 
47         /**
48          * @ORM\Column(type="decimal", precision=6, scale=2)
49          *
50          * @var float
51          */
52         private $value;
53 
54         /**
55          * @ORM\ManyToOne(targetEntity="User", inversedBy="courseCollection", cascade={\
56 "persist", "merge", "refresh"})
57          *
58          * @var User
59          */
60         protected $teacher;
61 
62         /**
63          * @ORM\OneToMany(targetEntity="Lesson", mappedBy="course", cascade={"all"}, or\
64 phanRemoval=true, fetch="LAZY")
65          *
66          * @var Doctrine\Common\Collections\Collection
67          */
68         protected $lessonCollection;
69 
70         /**
71          * @ORM\OneToMany(targetEntity="Enrollment", mappedBy="course", cascade={"all"}\
72 , orphanRemoval=true, fetch="LAZY")
73          */
74         private $enrollmentCollection;
75 
76         /**
77          * @return integer
78          */
79         public function getId()
80         {
81             return $this->id;
82         }
83 
84         public function getName()
```

```
85          {
86              return $this->name;
87          }
88
89          public function setName($name)
90          {
91              return $this->name = $name;
92          }
93
94          public function getDescription()
95          {
96              return $this->description;
97          }
98
99          public function setDescription($description)
100         {
101             return $this->description = $description;
102         }
103
104         public function getVideo()
105         {
106             return $this->video;
107         }
108
109         public function setVideo($video)
110         {
111             return $this->video = $video;
112         }
113
114         public function getPermalink()
115         {
116             return $this->permalink;
117         }
118
119         public function setPermalink($permalink)
120         {
121             return $this->permalink = $permalink;
122         }
123
124         public function getValue()
125         {
126             return $this->value;
```

```
127         }
128
129         public function setValue($value)
130         {
131             return $this->value = $value;
132         }
133
134         public function getTeacher()
135         {
136             return $this->teacher;
137         }
138
139         public function setTeacher($teacher)
140         {
141             return $this->teacher = $teacher;
142         }
143
144         public function getLessonCollection()
145         {
146             return $this->lessonCollection;
147         }
148
149         public function setLessonCollection($lessonCollection)
150         {
151             return $this->lessonCollection = $lessonCollection;
152         }
153
154         public function getEnrollmentCollection()
155         {
156             return $this->enrollmentCollection;
157         }
158
159         public function setEnrollmentCollection($enrollmentCollection)
160         {
161             return $this->enrollmentCollection = $enrollmentCollection;
162         }
163
164         public function __construct()
165         {
166         $this->lessonCollection = new ArrayCollection;
167         $this->enrollmentCollection = new ArrayCollection;
168     }
```

```
169
170 }
```

https://gist.github.com/eminetto/7312252[11]

Lesson

A seguir o código da classe *Lesson*, no arquivo *src/DoctrineNaPratica/Model/Lesson.php*

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="Lesson")
 */
class Lesson
{
        /**
         * @ORM\Id @ORM\Column(type="integer")
         * @ORM\GeneratedValue
         * @var integer
         */
        private $id;

        /**
         * @ORM\Column(type="string", length=150)
         *
         * @var string
         */
        private $name;

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

        /**
```

[11] https://gist.github.com/eminetto/7312252

```
34          * @ORM\Column(type="text", nullable=true)
35          *
36          * @var string
37          */
38         private $video;
39
40         /**
41          * @ORM\Column(type="string", length=255, nullable=true)
42          *
43          * @var string
44          */
45         private $permalink;
46
47         /**
48          * @ORM\ManyToOne(targetEntity="Course", inversedBy="lessonCollection", cascade\
49 ={"persist", "merge", "refresh"})
50          *
51          * @var Course
52          */
53         private $course;
54
55         /**
56      * @ORM\ManyToMany(targetEntity="User", mappedBy="lessonCollection")
57      */
58     private $userLessons;
59
60         /**
61          * @return integer
62          */
63         public function getId()
64         {
65             return $this->id;
66         }
67
68         public function getName()
69         {
70             return $this->name;
71         }
72
73         public function setName($name)
74         {
75             return $this->name = $name;
```

```
76          }
77
78          public function getDescription()
79          {
80              return $this->description;
81          }
82
83          public function setDescription($description)
84          {
85              return $this->description = $description;
86          }
87
88          public function getVideo()
89          {
90              return $this->video;
91          }
92
93          public function setVideo($video)
94          {
95              return $this->video = $video;
96          }
97
98          public function getCourse()
99          {
100             return $this->course;
101         }
102
103         public function setCourse($course)
104         {
105             return $this->course = $course;
106         }
107
108         public function getPermalink()
109         {
110             return $this->permalink;
111         }
112
113         public function setPermalink($permalink)
114         {
115             return $this->permalink = $permalink;
116         }
117
```

```
118         public function getUserLessons()
119         {
120         return $this->userLessons;
121         }
122
123         public function setUserLessons($userLessons)
124         {
125         return $this->userLessons = $userLessons;
126         }
127
128 }
```

https://gist.github.com/eminetto/7312259[12]

Enrollment

A seguir o código da classe *Enrollment*, no arquivo *src/DoctrineNaPratica/Model/Enrollment.php*

```
1  <?php
2  namespace DoctrineNaPratica\Model;
3
4  use Doctrine\ORM\Mapping as ORM;
5
6  /**
7   * @ORM\Entity
8   * @ORM\Table(name="Enrollment")
9   */
10 class Enrollment
11 {
12         /**
13          * @ORM\Id @ORM\Column(type="integer")
14          * @ORM\GeneratedValue
15          * @var integer
16          */
17         protected $id;
18
19         /**
20          * @ORM\ManyToOne(targetEntity="User", inversedBy="enrollmentCollection", casca\
21 de={"persist", "merge", "refresh"})
22          *
23          * @var User
24          */
```

[12] https://gist.github.com/eminetto/7312259

```
25          protected $user;
26
27          /**
28           * @ORM\ManyToOne(targetEntity="Course", inversedBy="enrollmentCollection", cas\
29  cade={"persist", "merge", "refresh"})
30           *
31           * @var Course
32           */
33          protected $course;
34
35          /**
36           * @ORM\Column(type="string", length=255, nullable=true)
37           *
38           * @var string
39           */
40          protected $certificationCode;
41
42          /**
43           * @return integer
44           */
45          public function getId()
46          {
47              return $this->id;
48          }
49
50
51          public function getUser()
52          {
53              return $this->user;
54          }
55
56          public function setUser($user)
57          {
58              $this->user = $user;
59          }
60
61          public function getCourse()
62          {
63              return $this->course;
64          }
65
66          public function setCourse($course)
```

```
67          {
68              $this->course = $course;
69          }
70
71          public function getCertificationCode()
72          {
73              return $this->certificationCode;
74          }
75
76          public function setCertificationCode($certificationCode)
77          {
78              $this->certificationCode = $certificationCode;
79          }
80  }
```

https://gist.github.com/eminetto/7312273[13]

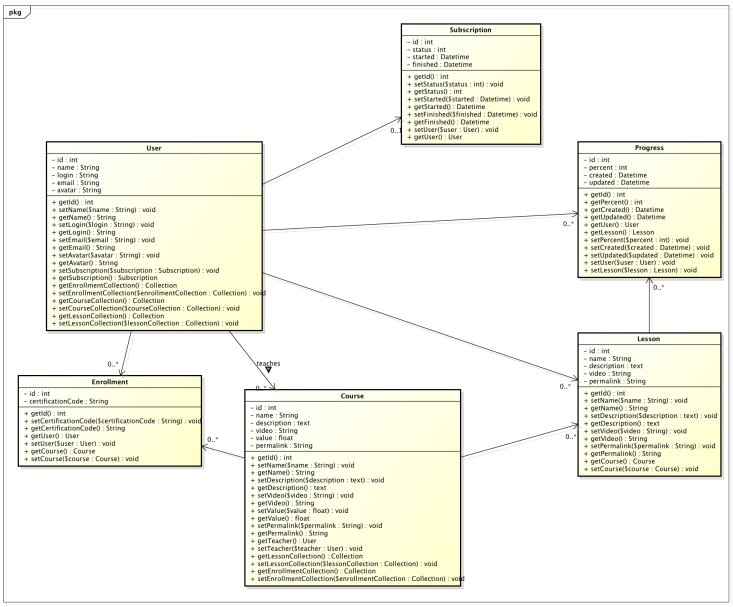
Vamos agora atualizar a base de dados com as novas tabelas e relações:

```
1  ./vendor/bin/doctrine orm:schema-tool:update --force
```

Progress

Vamos agora criar a entidade *Progress* que vai representar o progresso do aluno nas lições.

[13] https://gist.github.com/eminetto/7312273



Progress

A classe *Progress* deve ser criada no arquivo *src/DoctrineNaPratica/Model/Progress.php* e não apresenta novidades em relação ao que foi comentado anteriormente neste capítulo.

```
1  <?php
2
3  namespace DoctrineNaPratica\Model;
4
5  use Doctrine\ORM\Mapping as ORM;
6
7  /**
8   * @ORM\Entity
9   * @ORM\Table(name="Progress")
10  */
11 class Progress
12 {
13         /**
14          * @ORM\Id @ORM\Column(type="integer")
15          * @ORM\GeneratedValue
16          * @var integer
17          */
18         private $id;
19
20         /**
21          * @ORM\Column(type="integer")
22          *
```

```
23          * @var int
24          */
25         private $percent;
26
27         /**
28          * @ORM\ManyToOne(targetEntity="User", cascade={"persist", "merge", "refresh"})
29          *
30          * @var User
31          */
32         protected $user;
33
34         /**
35          * @ORM\ManyToOne(targetEntity="Lesson", cascade={"persist", "merge", "refresh"\
36 })
37          *
38          * @var Lesson
39          */
40         protected $lesson;
41
42         /**
43          * @ORM\Column(type="datetime")
44          *
45          * @var Datetime
46          */
47         protected $created;
48
49         /**
50          * @ORM\Column(type="datetime", nullable=true)
51          *
52          * @var Datetime
53          */
54         protected $updated;
55
56
57         /**
58          * @return integer
59          */
60         public function getId()
61         {
62             return $this->id;
63         }
64
```

```
65          public function getPercent()
66          {
67              return $this->percent;
68          }
69
70          public function setPercent($percent)
71          {
72              $this->percent = $percent;
73          }
74
75          public function getUser()
76          {
77              return $this->user;
78          }
79
80          public function setUser($user)
81          {
82              $this->user = $user;
83          }
84
85          public function getLesson()
86          {
87              return $this->lesson;
88          }
89
90          public function setLesson($lesson)
91          {
92              $this->lesson = $lesson;
93          }
94
95          public function getCreated()
96          {
97          return $this->created;
98          }
99
100         public function setCreated($created)
101         {
102         return $this->created = $created;
103         }
104
105         public function getUpdated()
106         {
```

```
107         return $this->updated;
108         }
109
110         public function setUpdated($updated)
111         {
112         return $this->updated = $updated;
113         }
114 }
```

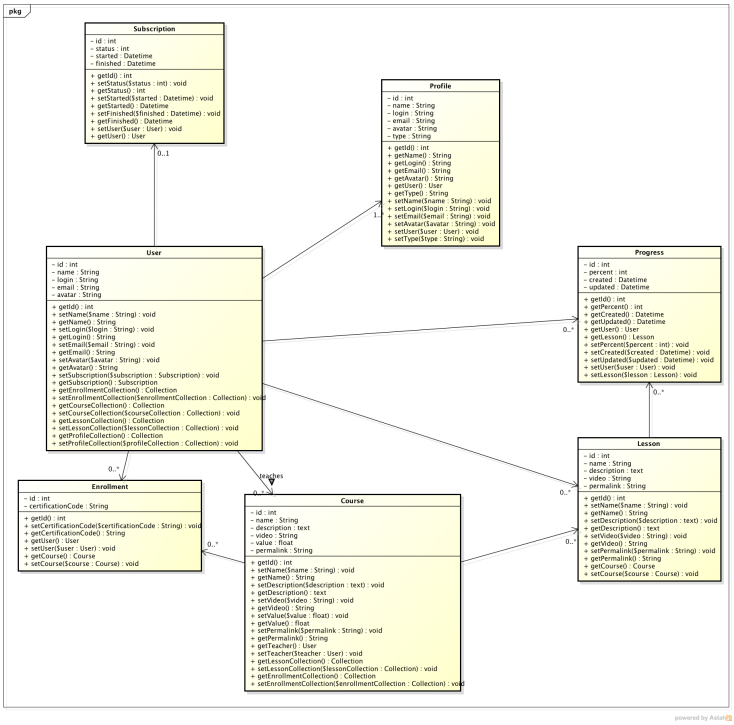
https://gist.github.com/eminetto/7312279[14]

Vamos atualizar a base de dados com as novas tabelas e relações:

```
1 ./vendor/bin/doctrine orm:schema-tool:update --force
```

Profiles

Vamos adicionar mais uma informação aos dados dos nossos alunos: os seus perfis sociais. Para isso vamos criar uma nova entidade, a *Profile* que vai nos trazer um novo formato de entidade. A seguir a nova modelagem do projeto.



Final

[14] https://gist.github.com/eminetto/7312279

Vamos começar adicionando uma nova relação na classe *User*, bem como alterando o construtor para adicionar a nova coleção de dados.

```
1  /**
2  * @ORM\OneToMany(targetEntity="Profile", mappedBy="user", cascade={"all"}, orpha\
3  nRemoval=true, fetch="LAZY")
4  *
5  * @var Doctrine\Common\Collections\Collection
6  */
7  protected $profileCollection;
8
9  public function getProfileCollection()
10 {
11         return $this->profileCollection;
12 }
13 public function setProfileCollection($profileCollection)
14 {
15         return $this->profileCollection = $profileCollection;
16 }
17
18 //alterar o construtor
19 public function __construct()
20 {
21         $this->courseCollection = new ArrayCollection;
22         $this->lessonCollection = new ArrayCollection;
23         $this->profileCollection = new ArrayCollection;
24         $this->enrollmentCollection = new ArrayCollection;
25 }
```

https://gist.github.com/eminetto/7312283[15]

Agora podemos criar a classe *Profile*, no arquivo *src/DoctrineNaPratica/Model/Profile.php*

[15] https://gist.github.com/eminetto/7312283

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
  * @ORM\Entity
  * @ORM\Table(name="Profile")
  * @ORM\InheritanceType("SINGLE_TABLE")
  * @ORM\DiscriminatorColumn(name="type", type="string")
  * @ORM\DiscriminatorMap({
  *                "twitter" = "TwitterProfile",
  *                "facebook" = "FacebookProfile",
  *                "github" = "GithubProfile"
  * })
  */
abstract class Profile
{
        /**
         * @ORM\Id @ORM\Column(type="integer")
         * @ORM\GeneratedValue
         * @var integer
         */
        private $id;

        /**
         * @ORM\Column(type="string")
         *
         * @var string
         */
        private $name;

        /**
         * @ORM\Column(type="string")
         *
         * @var string
         */
        private $login;

        /**
         * @ORM\Column(type="string")
         *
```

```
43          * @var string
44          */
45         private $email;
46
47         /**
48          * @ORM\Column(type="string", length=255, nullable=true)
49          *
50          * @var string
51          */
52         private $avatar;
53
54         /**
55          * @ORM\ManyToOne(targetEntity="User", inversedBy="profileCollection", cascade=\
56 {"persist", "merge", "refresh"})
57          *
58          * @var User
59          */
60         private $user;
61
62         /**
63          * @return integer
64          */
65         public function getId()
66         {
67             return $this->id;
68         }
69
70         public function getName()
71         {
72             return $this->name;
73         }
74
75         public function setName($name)
76         {
77             $this->name = $name;
78         }
79
80         public function getLogin()
81         {
82             return $this->login;
83         }
84
```

```
85          public function setLogin($login)
86          {
87              $this->login = $login;
88          }
89
90          public function getEmail()
91          {
92              return $this->email;
93          }
94
95          public function setEmail($email)
96          {
97              $this->email = $email;
98          }
99
100         public function getAvatar()
101         {
102             return $this->avatar;
103         }
104
105         public function setAvatar($avatar)
106         {
107             $this->avatar = $avatar;
108         }
109
110         public function getUser()
111         {
112             return $this->user;
113         }
114
115         public function setUser($user)
116         {
117             $this->user = $user;
118         }
119
120 }
```

https://gist.github.com/eminetto/7312286[16]

No nosso projeto vamos ter inicialmente três tipos possíveis de perfil social: Twitter, Facebook e Github. Mas como novas redes sociais surgem a cada instante precisamos de uma forma de deixar nosso projeto mais fácil de atender estes novos perfis no futuro. Para isso vamos definir nossa classe

[16] https://gist.github.com/eminetto/7312286

Perfil como uma classe abstrata que vai ser extendida e implementada pelas classes correspondentes as redes sociais que desejamos atender no projeto. Para isto estamos usando a anotação *@ORMInheritanceType* com o valor *SINGLE_TABLE* que indica ao Doctrine para criar apenas uma tabela na base de dados e usar o campo definido na anotação *@ORMDiscriminatorColumn* (o *type* no nosso caso) para distinguir os tipos de perfil. Os valores possíveis do campo *type* são os que definimos na anotação *@ORMDiscriminatorMap*, bem como o nome da classe correspondente, cujos códigos estão apresentados nas próximas páginas. Podemos também usar o valor *JOINED* para a anotação *@ORMInheritanceType*, o que indicaria ao Doctrine que ele deve criar uma tabela para cada classe filha da *Profile* e usar chaves estrangeiras para fazer a ligação. Cada formato tem suas vantagens e desvantagens, dependendo do caso e do projeto (número de registros, número de classes filhas, número de requisições esperadas, etc) e é recomendado analisar com calma a documentação oficial[17] antes de definir a melhor estratégia para cada caso.

Vamos agora definir o código das classes filhas de *Profile*

TwitterProfile

O código da classe *TwitterProfile*, no arquivo *src/DoctrineNaPratica/Model/TwitterProfile.php*

```
1  <?php
2  namespace DoctrineNaPratica\Model;
3
4  use Doctrine\ORM\Mapping as ORM;
5
6  /**
7    * @ORM\Entity
8    */
9  class TwitterProfile extends Profile
10 {
11 }
```

https://gist.github.com/eminetto/7312293[18]

FacebookProfile

O código da classe *FacebookProfile*, no arquivo *src/DoctrineNaPratica/Model/FacebookProfile.php*

[17] https://doctrine-orm.readthedocs.org/en/latest/reference/inheritance-mapping.html
[18] https://gist.github.com/eminetto/7312293

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
  * @ORM\Entity
  */
class FacebookProfile extends Profile
{
}
```

https://gist.github.com/eminetto/7312299[19]

GithubProfile

E o código da classe *GithubProfile*, no arquivo *src/DoctrineNaPratica/Model/GithubProfile.php*

```php
<?php
namespace DoctrineNaPratica\Model;

use Doctrine\ORM\Mapping as ORM;

/**
  * @ORM\Entity
  */
class GithubProfile extends Profile
{
}
```

https://gist.github.com/eminetto/7312305[20]

Atualmente as classes não possuem código específico, mas é fácil alterá-las para criar campos específicos para cada rede social que formos usar no projeto.

Agora que definimos as classes de nosso projeto, bem como a relação entre elas, vamos aprender como manipulá-las nos próximos capítulos.

[19] https://gist.github.com/eminetto/7312299
[20] https://gist.github.com/eminetto/7312305

Testes e manipulação de entidades

Introdução

O desenvolvimento orientado a testes (*TDD* na sua sigla em inglês) é uma técnica extremamente importante e algo que deve fazer parte da bagagem de conhecimentos de um programador PHP. O objetivo deste livro é apresentar o Doctrine e não o TDD ou o PHPUnit, ferramenta que vamos usar para implementar os testes, mas achei importante incluir este tópico devido a sua importância no ciclo de desenvolvimento.

Não vou aqui explicar todas as possibilidades de uso do PHPUnit pois isso é assunto para um livro a parte. Também não vou entrar na discussão do uso de *mocks* para emular o banco de dados, ou mesmo na real importância de testarmos a camada de dados de uma aplicação (alguns estudiosos defendem que devemos testar apenas os serviços e controladores) e vou usar os testes para demonstrar o uso do próprio Doctrine. Como não temos ainda uma aplicação com controladores ou mesmo interface para visualizarmos os dados nós vamos usar os testes unitários como guia no aprendizado da manipulação das nossas entidades pelo Doctrine.

Instalação e configuração

Vamos começar instalando o PHPUnit. Existe mais de uma forma de fazer a instalação, mas a mais fácil e rápida é usando o próprio *Composer*, como usamos anteriormente para instalar o Doctrine. Para isso basta atualizar nosso *composer.json*:

```
{
    "require": {
        "doctrine/common": "2.4.*",
        "doctrine/dbal": "2.4.*",
        "doctrine/orm": "2.4.*",
        "phpunit/phpunit": "3.7.*"
    }
}
```

E executar o Composer:

```
php composer.phar update
```

Precisamos criar um arquivo chamado *phpunit.xml*, na raiz do nosso projeto, com a configuração que o PHPUnit precisa para iniciar os testes:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
         bootstrap="tests/bootstrap.php"
>
    <testsuites>
        <testsuite name="Doctrine na prática Test Suite">
            <directory suffix=".php">tests/</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
            <directory suffix=".php">./src/</directory>
            <exclude>
                <directory suffix=".php">./vendor/</directory>
            </exclude>

        </whitelist>
    </filter>


    <!-- Code Coverage Configuration -->
    <logging>
        <log type="coverage-html" target="tests/_reports/coverage" charset="UTF-\
8" yui="true" highlight="true" lowUpperBound="35" highLowerBound="70" />
        <log type="testdox-text" target="tests/_reports/testdox/executed.txt"/>
    </logging>

</phpunit>
```

https://gist.github.com/eminetto/7363935[21]

Como todo arquivo *XML* ele é bem explicativo. Basicamente o que fazemos é definir qual é o diretório onde estão nossos arquivos (o *src*) e qual o diretório que ele não deve ler durante os testes (o *vendor*

[21] https://gist.github.com/eminetto/7363935

que contém os arquivos do próprio Doctrine e outros pacotes). As últimas linhas (o *logging*) são opcionais e podem ser removidas caso desejado. Elas servem para gerar relatórios dos resultados da execução dos nossos testes.

Precisamos também criar o diretório onde vamos criar nossos testes. No Linux/Mac OS X podemos fazer isso com o comando:

```
1  mkdir -p tests/src/DoctrineNaPratica/Model
```

Na linha 11 do *phpunit.xml* indicamos qual é o arquivo responsável pelo "bootstrap dos testes", o *tests/bootstrap.php*. Ele é responsável pela configuração básica necessária para os testes rodarem. Basicamente ele usa o bootstrap que criamos para a aplicação e muda algumas pequenas coisas:

```
1  <?php
2  require __DIR__ . '/../bootstrap.php';
3
4  use Doctrine\ORM\EntityManager;
5
6  // configurações do banco de dados
7  $dbParams = array(
8      'driver'   => 'pdo_sqlite',
9      'dbname'   => 'memory:dnp.db',
10 );
11
12 //cria o entityManager
13 return $entityManager = EntityManager::create($dbParams, $config);
```

https://gist.github.com/eminetto/7363945[22]

A principal mudança é a configuração do banco de dados. Na nossa aplicação normal estamos usando o *MySQL* e para os testes vamos usar o *SQLite*, pequeno e poderoso banco de dados embutido no PHP. O *SQLite* é muito mais rápido do que os outros bancos de dados e faz todo o sentido usarmos ele para nossos testes. Esta é uma das vantagens de usarmos uma ferramenta como o Doctrine que nos permite termos independência de bancos de dados. Além de usarmos o *SQLite* estamos configurando, na linha 9, para que o Doctrine use o banco em memória e não no sistema de arquivos, o que consegue aumentar ainda mais a performance dos nossos testes.

Criando o primeiro teste

Primeiro vamos criar uma classe base para todos os testes, a *TestCase*. Primeiro criamos o diretório para a classe:

[22] https://gist.github.com/eminetto/7363945

```
mkdir -p src/DoctrineNaPratica/Test
```

O conteúdo do arquivo *TestCase.php* é:

```
<?php

namespace DoctrineNaPratica\Test;

use Doctrine\ORM\Tools\SchemaTool;

abstract class TestCase extends \PHPUnit_Framework_TestCase
{
    /**
     * @var EntityManager
     */
    protected $em = null;

    public function setup()
    {
        $em = $this->getEntityManager();
        $tool = new SchemaTool($em);
        //busca a informação de todas as entidades
        $classes = $em->getMetadataFactory()->getAllMetadata();
        // cria a base de dados para os testes
        $tool->createSchema($classes);
        parent::setup();
    }

    public function tearDown()
    {
        $em = $this->getEntityManager();
        $tool = new SchemaTool($em);
        $classes = $em->getMetadataFactory()->getAllMetadata();
        // Remove a base de dados
        $tool->dropSchema($classes);
        parent::tearDown();
    }

    protected function getEntityManager()
    {
        if (! $this->em) {
            $this->em = require __DIR__ . '/../../../tests/bootstrap.php';
```

```
39          }
40          return $this->em;
41      }
42  }
```

https://gist.github.com/eminetto/7363966[23]

Se o leitor já teve algum contato com testes unitários usando o PHPUnit deve conhecer os métodos *setup* e *tearDown*. O *setup* é executado pelo PHPUnit antes de cada teste que escrevermos e é responsável pelas configurações necessárias para o teste rodar. O *tearDown* é executado após cada teste e sua função geralmente é a de limpar qualquer efeito que o teste tenha criado. A ideia destes dois métodos é garantir que cada teste seja realmente unitário, que um teste não influencie outro. No nosso caso, o método *setup* vai ser responsável por criar a base de dados antes de cada teste, e o *tearDown* vai remover a base após o fim do teste. Você pode optar também por apenas limpar a base de dados ao invés de removê-la, mas como estamos usando o *SQLite* este processo é muito rápido.

Nestes dois métodos podemos ver mais algumas funcionalidades interessantes do Doctrine, como a classe *SchemaTool*. Com ela podemos analisar as entidades que definimos anteriormente e manipulá-las em tempo de execução ou, como neste exemplo, podemos pedir para que ela crie ou remova todas as entidades da base de dados. O método *createSchema* é o equivalente ao comando que usamos anterioremente na linha de comando para gerar e atualizar a nossa base de dados. O *SchemaTool* pode ser bem útil para manipulações de entidades e modificações em tempo de execução.

Vamos agora criar a primeira versão dos nossos testes, o *tests/src/DoctrineNaPratica/Model/UserTest.php*:

```
1   <?php
2
3   namespace DoctrineNaPratica\Model;
4
5   use DoctrineNaPratica\Test\TestCase;
6   use DoctrineNaPratica\Model\User;
7
8   /**
9    * @group Model
10   */
11  class UserTest extends TestCase
12  {
13
14      public function testUser()
15      {
16          $user = new User;
```

[23] https://gist.github.com/eminetto/7363966

```
17          $user->setName('Steve Jobs');
18          $user->setLogin('steve');
19          $user->setLogin('steve@apple.com');
20          $user->setAvatar('steve.png');
21
22          $this->getEntityManager()->persist($user);
23          $this->getEntityManager()->flush();
24
25          //deve ter criado um id
26          $this->assertNotNull($user->getId());
27          $this->assertEquals(1, $user->getId());
28
29          $savedUser = $this->getEntityManager()->find(get_class($user), $user->ge\
30  tId());
31
32          $this->assertInstanceOf(get_class($user), $savedUser);
33          $this->assertEquals($user->getName(), $savedUser->getName());
34
35      }
36  }
```

https://gist.github.com/eminetto/7363971[24]

O primeiro teste serve para nos mostrar como o Doctrine realiza duas importantes manipulações das entidades: a criação e a leitura da base de dados.

Após criarmos uma nova instância da classe *User* e definirmos seus dados, da linha 16 a 20, vamos usar o componente do Doctrine chamado *EntityManager* para fazer a persistência do objeto. Ao usarmos o método *persist()* o *EntityManager* armazena o objeto em memória e ao invocarmos o *flush* este objeto é efetivamente salvo no banco de dados, de acordo com as configurações definidas no *bootstrap.php*. Podemos persistir diversos objetos (usando o *persist*) e só executar uma vez o comando *flush* para que o Doctrine faça apenas uma conexão ao banco de dados e salve as informações que estão em memória.

Na linha 29 estamos usando o *EntityManager* para recuperar um registro da base de dados e traduzí-lo para um objeto. O método *find* recebe dois parâmetros, sendo o primeiro a classe do objeto sendo manipulado (*DoctrineNaPraticaModelUser* no nosso caso) e o valor da chave primária a ser encontrado. O retorno vai ser um objeto do tipo solicitado ou *null*. Existem outras formas de recuperarmos o registro da base de dados, que veremos nos próximos capítulos.

As demais linhas do método *testUser* são invocações dos métodos do PHPUnit (os *assert*) e fazem parte da definição do nosso teste. A linha 27, por exemplo, diz para o PHPUnit que o valor esperado para o campo *id* do objeto é 1, ou seja, estamos testando se o código da chave primária está sendo

[24] https://gist.github.com/eminetto/7363971

gerado automaticamente, como definimos nas anotações da entidade. Os demais *assert* servem para testarmos os outros valores esperados. Mais detalhes sobre os métodos de asserção do PHPUnit podem ser encontrados na documentação oficial[25].

Agora podemos executar os testes pela primeira vez e verificar se estão funcionando:

```
./vendor/bin/phpunit
```

O resultado deve ser:

```
OK (1 test, 4 assertions)
```

Vamos agora adicionar mais testes, para verificarmos se as relações estão funcionando. Vamos também fazer uma pequena refatoração para facilitar a escrita dos testes, criando o método *buildUser*.

```
<?php

namespace DoctrineNaPratica\Model;

use DoctrineNaPratica\Test\TestCase;
use DoctrineNaPratica\Model\User;
use DoctrineNaPratica\Model\Subscription;
use DoctrineNaPratica\Model\Enrollment;
use DoctrineNaPratica\Model\Course;
use DoctrineNaPratica\Model\Lesson;
use DoctrineNaPratica\Model\GithubProfile;
use DoctrineNaPratica\Model\FacebookProfile;
use DoctrineNaPratica\Model\TwitterProfile;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @group Model
 */
class UserTest extends TestCase
{

    //testa a criação do User
    public function testUser()
    {
```

[25] http://phpunit.de/manual/3.7/pt_br/writing-tests-for-phpunit.html

```
25
26          $user = $this->buildUser();
27
28          $this->getEntityManager()->persist($user);
29          $this->getEntityManager()->flush();
30
31          //deve ter criado um id
32          $this->assertNotNull($user->getId());
33          $this->assertEquals(1, $user->getId());
34
35          $savedUser = $this->getEntityManager()->find(get_class($user), $user->ge\
36  tId());
37
38          $this->assertInstanceOf(get_class($user), $savedUser);
39          $this->assertEquals($user->getName(), $savedUser->getName());
40      }
41
42      //testa a Subscription do User
43      public function testUserSubscription()
44      {
45          $user = $this->buildUser();
46
47          $subscription = new Subscription;
48          $subscription->setStatus(1);
49          $subscription->setStarted(new \DateTime('NOW'));
50
51          $user->setSubscription($subscription);
52          $this->getEntityManager()->persist($user);
53          $this->getEntityManager()->flush();
54
55          //deve ter criado um id
56          $this->assertNotNull($subscription->getId());
57          $this->assertEquals(1, $subscription->getId());
58
59          $savedUser = $this->getEntityManager()->find(get_class($user), $user->ge\
60  tId());
61
62          $this->assertInstanceOf(get_class($subscription), $savedUser->getSubscri\
63  ption());
64          $this->assertEquals($subscription->getId(), $savedUser->getSubscription(\
65  )->getId());
66      }
```

```
67
68      //testa os Enrollments do User
69      public function testUserEnrollment()
70      {
71          $user = $this->buildUser();
72          $courseA = $this->buildCourse('PHP');
73          $courseB = $this->buildCourse('Doctrine');
74
75          $enrollmentA = new Enrollment;
76          $enrollmentA->setUser($user);
77          $enrollmentA->setCourse($courseA);
78
79          $enrollmentB = new Enrollment;
80          $enrollmentB->setUser($user);
81          $enrollmentB->setCourse($courseB);
82
83          $enrollmentCollection = new ArrayCollection();
84          $enrollmentCollection->add($enrollmentA);
85          $enrollmentCollection->add($enrollmentB);
86
87          $user->setEnrollmentCollection($enrollmentCollection);
88
89          $this->getEntityManager()->persist($user);
90          $this->getEntityManager()->flush();
91
92          $savedUser = $this->getEntityManager()->find(get_class($user), 1);
93          $this->assertEquals(2, count($savedUser->getEnrollmentCollection()));
94      }
95
96      //testa os Courses do User (teacher)
97      public function testUserCourse()
98      {
99          $user = $this->buildUser();
100
101         $courseA = new Course;
102         $courseA->setName('PHP');
103         $courseA->setDescription('Curso de PHP');
104         $courseA->setValue(100);
105         $courseA->setTeacher($user);
106
107         $courseB = new Course;
108         $courseB->setName('Doctrine');
```

```
109         $courseB->setDescription('Curso de Doctrine');
110         $courseB->setValue(400);
111         $courseB->setTeacher($user);
112 
113         $courseCollection = new ArrayCollection;
114         $courseCollection->add($courseA);
115         $courseCollection->add($courseB);
116         $user->setCourseCollection($courseCollection);
117 
118         $this->getEntityManager()->persist($user);
119         $this->getEntityManager()->flush();
120 
121         $savedUser = $this->getEntityManager()->find(get_class($user), 1);
122         $this->assertEquals(2, count($savedUser->getCourseCollection()));
123         $savedCourses = $savedUser->getCourseCollection();
124         $this->assertEquals('Doctrine', $savedCourses[1]->getName());
125     }
126 
127     //testa as Lessons do User
128     public function testUserLesson()
129     {
130         $user = $this->buildUser();
131 
132         $course = new Course;
133         $course->setName('PHP');
134         $course->setDescription('Curso de PHP');
135         $course->setValue(100);
136         $course->setTeacher($user);
137 
138         $lessonA = new Lesson;
139         $lessonA->setName('Arrays');
140         $lessonA->setDescription('Aula sobre Arrays');
141         $lessonA->setCourse($course);
142 
143         $lessonB = new Lesson;
144         $lessonB->setName('Datas');
145         $lessonB->setDescription('Aula sobre Datas');
146         $lessonB->setCourse($course);
147 
148         $lessonCollection = new ArrayCollection;
149         $lessonCollection->add($lessonA);
150         $lessonCollection->add($lessonB);
```

```
151         $user->setLessonCollection($lessonCollection);
152 
153         $this->getEntityManager()->persist($user);
154         $this->getEntityManager()->flush();
155 
156         $savedUser = $this->getEntityManager()->find(get_class($user), 1);
157         $this->assertEquals(2, count($savedUser->getLessonCollection()));
158         $savedLessons = $savedUser->getLessonCollection();
159         $this->assertEquals(1, $savedLessons[0]->getId());
160     }
161 
162     //testa as Profiles do User
163     public function testUserProfile()
164     {
165         $user = $this->buildUser();
166 
167         $github = new GithubProfile;
168         $github->setName('Elton Minetto');
169         $github->setLogin('eminetto');
170         $github->setEmail('eminetto@coderockr.com');
171         $github->setAvatar('eminetto.png');
172 
173         $facebook = new FacebookProfile;
174         $facebook->setName('Elton Luís Minetto');
175         $facebook->setLogin('eminetto');
176         $facebook->setEmail('eminetto@gmail.com');
177         $facebook->setAvatar('eminetto.png');
178 
179         $twitter = new TwitterProfile;
180         $twitter->setName('Elton Minetto');
181         $twitter->setLogin('eminetto');
182         $twitter->setEmail('eminetto@coderockr.com');
183         $twitter->setAvatar('eminetto.jpg');
184 
185         $profileCollection = new ArrayCollection;
186         $profileCollection->add($github);
187         $profileCollection->add($facebook);
188         $profileCollection->add($twitter);
189         $user->setProfileCollection($profileCollection);
190 
191         $this->getEntityManager()->persist($user);
192         $this->getEntityManager()->flush();
```

```
193
194         $savedUser = $this->getEntityManager()->find(get_class($user), 1);
195         $this->assertEquals(3, count($savedUser->getProfileCollection()));
196         $savedProfiles = $savedUser->getProfileCollection();
197
198         $this->assertInstanceOf(get_class($github), $savedProfiles[0]);
199         $this->assertInstanceOf(get_class($facebook), $savedProfiles[1]);
200         $this->assertInstanceOf(get_class($twitter), $savedProfiles[2]);
201     }
202
203     //cria um User
204     private function buildUser()
205     {
206         $user = new User;
207         $user->setName('Steve Jobs');
208         $user->setLogin('steve');
209         $user->setLogin('steve@apple.com');
210         $user->setAvatar('steve.png');
211
212         return $user;
213     }
214
215     //cria um Course
216     private function buildCourse($courseName)
217     {
218         $teacher = $this->buildUser();
219         //login é unique
220         $teacher->setLogin('jobs'.$courseName);
221
222         $course = new Course;
223         $course->setName($courseName);
224         $course->setDescription('Curso de PHP');
225         $course->setValue(100);
226         $course->setTeacher($teacher);
227
228         return $course;
229     }
230 }
```

https://gist.github.com/eminetto/7363976[26]

[26] https://gist.github.com/eminetto/7363976

Nos novos testes criados podemos ver a criação e recuperação das relações entre as entidades. Na linha 47, por exemplo, estamos criando uma nova instância de *Subscription* e após definirmos seus dados fazemos a associação do usuário com este objeto, na linha 51. Ao fazermos o *persist* do *_User*, na linha 52, seguido de um *flush* o Doctrine verifica na memória que existe essa associação e faz o salvamento de ambos os registros nas respectivas tabelas do banco de dados. Na linha 62 podemos ver o uso do método *getSubscription()* para que tenhamos acesso ao objeto correspondente ao registro da base de dados. Esta é uma das vantagens do Doctrine, pois ao recuperarmos o objeto *User* da base de dados podemos ter acesso facilmente a todos os registros que estão vinculados a ele, como as suas lições (linha 157).

Outro ponto interessante consta nas linhas 158 e 159 do arquivo. Ao usarmos o método *getLessonCollection()* o Doctrine vai nos retornar um objeto do tipo *ArrayCollection* que possui todas as instâncias de *Lesson* associadas aquele usuário. Apesar de ser um objeto, o *ArrayCollection* nos permite acessar seu conteúdo na forma de um array (como mostrado na linha 159) além de possuir diversos outros métodos próprios.

Rodando os testes novamente é possível ver o resultado dos novos testes:

./vendor/bin/phpunit

```
1  OK (6 tests, 17 assertions)
```

No decorrer do arquivo de testes fizemos a manipulação de todas as entidades relacionadas ao objeto *User*. Usando o *UserTest.php* como exemplo é possível criarmos os testes para as demais classes. Deixo esta tarefa como exercício e desafio para o leitor.

Recuperação e manipulação de dados

No capítulo anterior vimos a forma mais básica de recupear dados usando o *EntityManager*. Neste capítulo vamos aprender a criar queries mais complexas.

DBAL

Conforme vimos no primeiro capítulo, o *DBAL* é o componente básico do Doctrine e serve como uma camada de abstração entre as bibliotecas *PDO* e o restante do framework, como o próprio *ORM*. Nós podemos usar apenas o *DBAL* para manipular a nossa base de dados caso não desejemos usar todo o *ORM* e os outros componentes do Doctrine.

No teste a seguir inclui diversos exemplos de como usar o *DBAL* para realizar algumas operações básicas. Vamos criar o arquivo *tests/src/DoctrineNaPratica/Model/DbalTest.php* com o conteúdo:

```
1  <?php
2
3  namespace DoctrineNaPratica\Model;
4
5  use DoctrineNaPratica\Test\TestCase;
6  use Doctrine\DBAL\DriverManager;
7  use Doctrine\DBAL\Schema\Schema;
8
9  /**
10  * @group Model
11  */
12 class DbalTest extends TestCase
13 {
14
15     private $conn;
16     private $schema;
17
18     public function setup()
19     {
20         $connectionParams = array(
21             'dbname'    => 'sqlite:memory',
22             'driver'    => 'pdo_sqlite',
23         );
```

```
24          //conectar na base
25          $this->conn = DriverManager::getConnection($connectionParams);
26          //cria uma nova base de dados e uma tabela
27          $this->schema = new Schema();
28          $usersTable = $this->schema->createTable("users");
29          $usersTable->addColumn("id", "integer", array("unsigned" => true));
30          $usersTable->addColumn("name", "string", array("length" => 100));
31          $usersTable->addColumn("login", "string", array("length" => 100));
32          $usersTable->addColumn("email", "string", array("length" => 256));
33          $usersTable->addColumn("avatar", "string", array("length" => 256));
34          $usersTable->setPrimaryKey(array("id"));
35          //busca a plataforma de banco de dados configurada
36          $platform = $this->conn->getDatabasePlatform();
37          $queries = $this->schema->toSql($platform);
38          foreach ($queries as $q) {
39              $stmt = $this->conn->query($q);
40          }
41      }
42
43      public function tearDown()
44      {
45          $this->schema->dropTable("users");
46      }
47
48
49      //testa a consulta dos Users
50      public function testUser()
51      {
52          //inicia uma transação
53          $this->conn->beginTransaction();
54          $this->conn->insert('users', array(
55              'name' => 'Steve Jobs', 'login' => 'steve', 'email' => 'steve@apple.\
56  com', 'avatar' => 'steve.png'
57          ));
58
59          $this->conn->insert('users', array(
60              'name' => 'Bill Gates', 'login' => 'bill', 'email' => 'bill@microsof\
61  t.com', 'avatar' => 'bill.png'
62          ));
63          //commit da transação
64          $this->conn->commit();
65
```

```
66          $sql = "select * from users";
67          $stmt = $this->conn->query($sql);
68          $result = $stmt->fetchAll();
69          $this->assertEquals(2, count($result));
70          $this->assertEquals('steve', $result[0]['login']);
71          $this->assertEquals('bill', $result[1]['login']);
72      }
73
74      //testa a consulta dos Users com parametros
75      public function testUserParameters()
76      {
77          //inicia uma transação
78          $this->conn->beginTransaction();
79          $this->conn->insert('users', array(
80              'name' => 'Steve Jobs', 'login' => 'steve', 'email' => 'steve@apple.\
81  com', 'avatar' => 'steve.png'
82          ));
83
84          $this->conn->insert('users', array(
85              'name' => 'Bill Gates', 'login' => 'bill', 'email' => 'bill@microsof\
86  t.com', 'avatar' => 'bill.png'
87          ));
88          //commit da transação
89          $this->conn->commit();
90
91          //usando placeholders
92          $sql = 'SELECT * FROM users u WHERE u.login = ? or u.email = ?';
93          $stmt = $this->conn->prepare($sql);
94          $stmt->bindValue(1, 'steve');
95          $stmt->bindValue(2, 'steve@apple.com');
96          $stmt->execute();
97          $result = $stmt->fetchAll();
98          $this->assertEquals(1, count($result));
99          $this->assertEquals('steve', $result[0]['login']);
100
101         //usando Named parameters
102         $sql = 'SELECT * FROM users u WHERE u.login = :login or u.email = :email\
103 ';
104         $stmt = $this->conn->prepare($sql);
105         $stmt->bindValue('login', 'steve');
106         $stmt->bindValue('email', 'steve@apple.com');
107         $stmt->execute();
```

```
108         $result = $stmt->fetchAll();
109         $this->assertEquals(1, count($result));
110         $this->assertEquals('steve', $result[0]['login']);
111     }
112 }
```

https://gist.github.com/eminetto/7363984[27]

Nas linhas 6 a 7 do *DbalTest* nós fazemos a importação das classes necessárias para o uso do *DBAL*.

O primeiro passo a fazermos é a conexão com a base de dados, que é realizada pela classe *DriverManager* na linha 25. No trecho de código entre as linhas 27 e 40 é possível ver o uso da classe *Schema* para criarmos uma tabela na base de dados, definindo suas colunas e chave primária. Isto é útil para podermos criar ou modificar tabelas em tempo de execução ou para criar scripts de deploy e instalação de sistemas. E no método *tearDown* usamos a mesma classe para remover a tabela criada anteriormente.

No método *testUser* vamos aprender bastante. Iniciando pela linha 53, onde criamos uma transação usando o método *beginTransaction*. Este método é importante para podermos marcar o início das transações que queremos garantir que sejam realizadas no banco de dados. Tudo que for executado pelo *DBAL* após o *beginTransaction* só será efetivado no banco de dados quando invocarmos o método *commit* (linha 64). Isto é útil para podermos desfazer alguma transação que não foi realizada com sucesso ou o usuário escolheu por interrompê-la antes do fim. Nas linhas 54 e 59 vemos um exemplo de uso do comando *insert* para criar novos registros na tabela *user*, passando como parâmetro um array de dados a serem salvos. E entre as linhas 66 e 68 podemos ver a execução de uma consulta *SQL* para retornar os dados da base de dados. Os dados são retornados na forma de um array conforme pode ser visto nas linhas 70 e 71.

O método *testUserParameters* começa bem parecido com o anterior, criando uma transação e inserindo dois registros na base de dados usando o método *insert*. A grande diferença aqui é na forma como estamos criando as consultas *SQL* para retornar os dados, usando parâmetros. Com o *DBAL* podemos fazer uma consulta usando parâmetros de duas formas: usando *placeholders* ou usando *named parameters*. Na linha 92 definimos dois valores a serem substituídos (os pontos de interrogação) e nas linhas 94 e 95 estamos indicando quais são os valores a serem usados na consulta. A diferença entre esta abordagem e a mostrada entre as linhas 102 e 105 é que com os *named parameters* fica mais fácil a manutenção do sistema, por ser mais simples identificar quais valores estão sendo usados na consulta. Usar uma destas abordagens é uma boa prática pois melhora a performance das consultas por usar o cache de consultas que a grande maioria dos sistemas gerenciadores de bancos de dados implementa.

Outros métodos úteis do *DBAL* não mostrados nos testes são:

update

[27] https://gist.github.com/eminetto/7363984

```php
$this->conn->update('user', array('login' => 'bob'), array('id' => 1));
```

delete

```php
$this->conn->delete('user', array('id' => 1));
```

Mais exemplos podem ser vistos na documentação oficial[28].

Doctrine Query Language

A *Doctrine Query Language*, ou apenas *DQL*, é uma linguagem que nos permite fazer consultas em nossas entidades. Diferente do *DBAL* que é baseado em *SQL*, com a *DQL* nós vamos fazer consultas em nossas classes e vamos receber como retorno coleções de objetos. Uma boa analogia é pensarmos que nossas classes de entidades estão armazenadas em um repositório, uma base de objetos, e vamos usar *DQL* para recuperá-los. Por isso não vamos usar os nomes de tabelas e campos nas consultas e sim os nomes das classes e suas relações, conforme definimos nos capítulos anteriores.

Exemplos de queries

Vamos ver alguns exemplos de consultas que podemos fazer com *DQL*.

```php
//buscar todos os usuários
$query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User u');
$users = $query->getResult();

//buscar lições que tenham 100 de progresso
$query = $this->em->createQuery('SELECT p FROM DoctrineNapratica\Model\Progress \
p JOIN p.lesson l JOIN l.course c WHERE p.percent = 100');
$result = $query->getResult();

//buscar usuários usando parâmetros
$query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User u WH\
ERE u.login = :login');
$query->setParameter('login', 'Bob');
$users = $query->getResult();

//buscar usuários usando múltiplos parâmetros
$query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User u WH\
ERE u.login = :login or u.email = :email');
```

[28] http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/data-retrieval-and-manipulation.html#update

```
19 $query->setParameters(
20     array('login' => 'userA', 'email' => 'userA@domain.com')
21 );
22 $users = $query->getResult();
```

https://gist.github.com/eminetto/7363986[29]

A primeira consulta é bem simples e ajuda a vermos a difença entre uma consulta *SQL* e uma *DQL* pois estamos selecionando dados da classe *User*.

A segunda consulta introduz o uso de duas *JOIN* para buscarmos os dados das entidades relacionadas à classe *User*. Perceba que estamos usando apenas o nome do atributo da classe que faz a ligação entre as entidades (o *p.lesson* por exemplo). O Doctrine vai ler as anotações da classe e traduzir isso em uma consulta *SQL* que faça a ligação entre as tabelas da base de dados.

As duas consultas restantes mostram o uso de parâmetros para realizarmos as consultas, bem parecido com o formato usado pelo *DBAL* que vimos anteriormente.

Testando

No teste a seguir fazemos uso do *EntityManager* e do *DQL* para criar e recupar as entidades, conforme os exemplos anteriores.

```
1  <?php
2  //tests/src/DoctrineNaPratica/Model/QdlTest.php
3
4  namespace DoctrineNaPratica\Model;
5
6  use DoctrineNaPratica\Test\TestCase;
7  use DoctrineNaPratica\Model\User;
8  use DoctrineNaPratica\Model\Course;
9  use DoctrineNaPratica\Model\Lesson;
10 use DoctrineNaPratica\Model\Progress;
11 use DoctrineNaPratica\Model\Enrollment;
12 use Doctrine\Common\Collections\ArrayCollection;
13
14 /**
15  * @group Model
16  */
17 class DqlTest extends TestCase
18 {
19
```

[29] https://gist.github.com/eminetto/7363986

```
20      //testa a dql dos Users
21      public function testUser()
22      {
23          $userA = $this->buildUser('userA');
24          $userB = $this->buildUser('userB');
25
26          $this->getEntityManager()->persist($userA);
27          $this->getEntityManager()->persist($userB);
28          $this->getEntityManager()->flush();
29
30          $query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\U\
31  ser u');
32          $users = $query->getResult();
33          $this->assertEquals(2, count($users));
34
35          $this->assertEquals($userA->getId(), $users[0]->getId());
36          $this->assertEquals($userB->getId(), $users[1]->getId());
37
38          $this->assertInstanceOf(get_class($userA), $users[0]);
39      }
40
41
42      //buscar lições que tenham 100 de progresso
43      public function testUserLesson()
44      {
45          $userA = $this->buildUser('UserA');
46
47          $course = new Course;
48          $course->setName('PHP');
49          $course->setDescription('Curso de PHP');
50          $course->setValue(100);
51          $course->setTeacher($userA);
52
53          $lessonA = new Lesson;
54          $lessonA->setName('Arrays');
55          $lessonA->setDescription('Aula sobre Arrays');
56          $lessonA->setCourse($course);
57
58          $lessonB = new Lesson;
59          $lessonB->setName('Datas');
60          $lessonB->setDescription('Aula sobre Datas');
61          $lessonB->setCourse($course);
```

```
62
63          $lessonCollection = new ArrayCollection;
64          $lessonCollection->add($lessonA);
65          $lessonCollection->add($lessonB);
66          $userA->setLessonCollection($lessonCollection);
67
68          $enrollment = new Enrollment;
69          $enrollment->setUser($userA);
70          $enrollment->setCourse($course);
71
72          $progressA = new Progress;
73          $progressA->setPercent(100);
74          $progressA->setUser($userA);
75          $progressA->setLesson($lessonA);
76          $progressA->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
77  Y-m-d H:i:s')));
78
79          $progressB = new Progress;
80          $progressB->setPercent(90);
81          $progressB->setUser($userA);
82          $progressB->setLesson($lessonB);
83          $progressB->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
84  Y-m-d H:i:s')));
85
86          $this->getEntityManager()->persist($userA);
87          $this->getEntityManager()->persist($enrollment);
88          $this->getEntityManager()->persist($progressA);
89          $this->getEntityManager()->persist($progressB);
90          $this->getEntityManager()->flush();
91
92          $query = $this->em->createQuery(
93              'SELECT p FROM DoctrineNapratica\Model\Progress p JOIN p.lesson l JO\
94  IN l.course c WHERE p.percent = 100'
95          );
96          $result = $query->getResult();
97
98          $this->assertEquals(1, count($result));
99
100         $this->assertEquals($userA->getId(), $result[0]->getUser()->getId());
101         $this->assertEquals($lessonA->getId(), $result[0]->getLesson()->getId());
102     }
103
```

```
104     //testa a dql dos Users com parametros
105     public function testUserParameters()
106     {
107         $userA = $this->buildUser('userA');
108         $userB = $this->buildUser('userB');
109 
110         $this->getEntityManager()->persist($userA);
111         $this->getEntityManager()->persist($userB);
112         $this->getEntityManager()->flush();
113 
114         $query = $this->em->createQuery(
115             'SELECT u FROM DoctrineNapratica\Model\User u WHERE u.login = :login\
116  or u.email = :email'
117         );
118         $query->setParameters(
119             array('login' => 'userA', 'email' => 'userA@domain.com')
120         );
121         $users = $query->getResult();
122 
123         $this->assertEquals(1, count($users));
124 
125         $this->assertEquals($userA->getId(), $users[0]->getId());
126 
127         $this->assertInstanceOf(get_class($userA), $users[0]);
128     }
129 
130     //cria um User
131     private function buildUser($login)
132     {
133         $user = new User;
134         $user->setName($login . ' name');
135         $user->setLogin($login);
136         $user->setEmail($login . '@domain.com');
137         $user->setAvatar($login . '.png');
138 
139         return $user;
140     }
141 
142     //cria um Course
143     private function buildCourse($courseName)
144     {
145         $teacher = $this->buildUser();
```

```
146         //login é unique
147         $teacher->setLogin('jobs'.$courseName);
148 
149         $course = new Course;
150         $course->setName($courseName);
151         $course->setDescription('Curso de PHP');
152         $course->setValue(100);
153         $course->setTeacher($teacher);
154 
155         return $course;
156     }
157 }
```

https://gist.github.com/eminetto/7363989[30]

Executando o comando abaixo é possível ver se os testes passaram com sucesso.

```
1 ./vendor/bin/phpunit
```

Hydrators

Como vimos nos exemplos acima e nos testes, as consultas *DQL* retornam coleções de objetos. Podemos mudar este comportamento usando um recurso chamado *hydrators*.

Existem quatro opções de *hydrators* para usarmos:

- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR

Vamos adicionar novos testes para demonstrar o uso dos tipos de *hydrators*.

[30] https://gist.github.com/eminetto/7363989

```php
//testa os hydrators
public function testHydrators()
{
    $userA = $this->buildUser('userA');

    $this->getEntityManager()->persist($userA);
    $this->getEntityManager()->flush();

    //retorna os dados no formato de um array de objetos (é o padrão)
    $query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User \
u');
    $users = $query->getResult(\Doctrine\ORM\Query::HYDRATE_OBJECT);
    $this->assertInstanceOf(get_class($userA), $users[0]);
    $this->assertEquals($userA->getId(), $users[0]->getId());

    //retorna os dados na forma de um array de arrays
    $query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User \
u');
    $users = $query->getResult(\Doctrine\ORM\Query::HYDRATE_ARRAY);
    $this->assertTrue(is_array($users[0]));
    $this->assertEquals($userA->getId(), $users[0]['id']);

    //O alias da entidade é adicionado ao indice do array.
    $query = $this->em->createQuery('SELECT u FROM DoctrineNapratica\Model\User \
u');
    $users = $query->getResult(\Doctrine\ORM\Query::HYDRATE_SCALAR);
    $this->assertTrue(is_array($users[0]));
    $this->assertEquals($userA->getId(), $users[0]['u_id']);

    //se a consulta retorna apenas um elemento podemos usar o single_scalar
    $query = $this->em->createQuery('SELECT u.login FROM DoctrineNapratica\Model\
\User u');
    $login = $query->getResult(\Doctrine\ORM\Query::HYDRATE_SINGLE_SCALAR);
    $this->assertEquals($userA->getLogin(), $login);
}
```

https://gist.github.com/eminetto/7363994[31]

Na linha 12 do método *testHydrators* estamos executando a consulta e indicando que queremos o resultado na forma de objetos, usando o parâmetro *DoctrineORMQuery::HYDRATE_OBJECT*. Como podemos ver nas próximas duas linhas o resultado é uma instância da classe *User*. Este é o comportamento padrão do Doctrine.

[31] https://gist.github.com/eminetto/7363994

Os *hydrators* *DoctrineORMQuery::HYDRATE_ARRAY* e *DoctrineORMQuery::HYDRATE_SCALAR* tem efeitos parecidos pois ambos indicam ao Doctrine que o resultado desejado deve ser formatado como um array. A diferença é que o segundo vai adicionar o apelido da entidade ao índice do array. Podemos ver o uso destes dois *hydrators* entre as linhas 19 e 28 do trecho de código acima.

O último *hydrator* apresentado é o *DoctrineORMQuery::HYDRATE_SINGLE_SCALAR* que pode ser usado quando esperamos que nossa consulta retorne apenas um resultado, como o apresentado na linha 31. Isto é útil quando precisamos fazer alguma contagem ou somatório com os dados vindos do banco de dados e usar o *HYDRATE_SINGLE_SCALAR* nestes casos significa economia de recursos pois o resultado consome menos memória.

O QueryBuilder

Como estamos trabalhando com representações dos nossos dados na forma de objetos o mais recomendado é darmos preferência ao uso da *DQL* para a manipulação das informações dos nossos projetos. Para facilitar a criação de consultas *DQL* dinâmicas e mais efetivas o Doctrine fornece um componente chamado *QueryBuilder* que nos ajuda nesta tarefa.

Exemplos de queries

Vamos ver alguns exemplos de como usar o *QueryBuilder*:

```
1  //buscar todos os usuários
2  $qb = $this->em->createQueryBuilder();
3  $qb->select('u')->from('DoctrineNapratica\Model\User', 'u');
4  $query = $qb->getQuery();
5  $users = $query->getResult();
6
7  //buscar lições que tenham 100 de progresso
8  $qb = $this->em->createQueryBuilder();
9  $qb->select('p')
10    ->from('DoctrineNapratica\Model\Progress', 'p')
11    ->innerJoin('p.lesson', 'l')
12    ->innerJoin('l.course', 'c')
13    ->where('p.percent = 100');
14 $query = $qb->getQuery();
15 $users = $query->getResult();
16
17 //buscar usuários usando parâmetros
18 $qb = $this->em->createQueryBuilder();
19 $qb->select('u')
20    ->from('DoctrineNapratica\Model\User', 'u')
```

```
21      ->where('u.login = :login or u.email = :email')
22      ->setParameters(array('login' => 'userA', 'email' => 'userA@domain.com'));
23  $query = $qb->getQuery();
24  $users = $query->getResult();
```

https://gist.github.com/eminetto/7363999[32]

A primeira tarefa é criarmos uma instância do *QueryBuilder*, o que pode ser facilitado usando o próprio *EntityManager* para nos retornar uma, conforme pode ser viso na linha 2 do trecho de código acima.

O objeto *QueryBuilder* possui métodos que nos ajudam a criar as consultas, como *select*, *from*, *innerJoin* e *where*. Outro método importante é o *setParameters* com o qual podemos criar consultas com parâmetros nomeados, da mesma forma que vimos nos exemplos anteriores deste capítulo.

Uma das vantagens de se usar o *QueryBuilder* é podermos criar consultas dinâmicas, de acordo com ações do usuário ou do próprio sistema, como no exemplo a seguir:

```
1  $qb = $this->entityManager->createQueryBuilder();
2  $qb->select('category')
3      ->from('Sample\Model\Category', 'category');
4
5  if ($category_id) {
6      $where = 'category.parentCategory in (:category) ';
7      $parameters['category'] = $category_id;
8      $qb->where($where)->setParameters($parameters);
9  }
10 else {
11     $where = 'category.parentCategory is not null ';
12     $qb->where($where);
13 }
14 $query = $qb->getQuery();
15 $subCategories = $query->getResult();
```

Neste exemplo, caso o usuário envie pela *URL* o código da categoria nós adicionamos este código na cláusula *where* da consulta. Caso contrário retornamos todas as categorias pai.

Testando

Vamos continuar nossos testes, agora criando mais exemplos de consultas criadas pelo *QueryBuilder* criando o arquivo *tests/src/DoctrineNaPratica/Model/QbTest.php*.

[32] https://gist.github.com/eminetto/7363999

```php
<?php
namespace DoctrineNaPratica\Model;

use DoctrineNaPratica\Test\TestCase;
use DoctrineNaPratica\Model\User;
use DoctrineNaPratica\Model\Course;
use DoctrineNaPratica\Model\Lesson;
use DoctrineNaPratica\Model\Progress;
use DoctrineNaPratica\Model\Enrollment;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @group Model
 */
class QBTest extends TestCase
{

    //busca todos os usuários
    public function testUser()
    {
        $userA = $this->buildUser('userA');
        $userB = $this->buildUser('userB');

        $this->getEntityManager()->persist($userA);
        $this->getEntityManager()->persist($userB);
        $this->getEntityManager()->flush();

        $qb = $this->em->createQueryBuilder();
        $this->assertEquals('Doctrine\ORM\QueryBuilder', get_class($qb));

        $qb->select('u')
           ->from('DoctrineNapratica\Model\User', 'u');

        $query = $qb->getQuery();
        $users = $query->getResult();
        $this->assertEquals(2, count($users));

        $this->assertEquals($userA->getId(), $users[0]->getId());
        $this->assertEquals($userB->getId(), $users[1]->getId());

        $this->assertInstanceOf(get_class($userA), $users[0]);
    }
```

```
43
44
45      //buscar lições que tenham 100 de progresso
46      public function testUserLesson()
47      {
48          $userA = $this->buildUser('UserA');
49
50          $course = new Course;
51          $course->setName('PHP');
52          $course->setDescription('Curso de PHP');
53          $course->setValue(100);
54          $course->setTeacher($userA);
55
56          $lessonA = new Lesson;
57          $lessonA->setName('Arrays');
58          $lessonA->setDescription('Aula sobre Arrays');
59          $lessonA->setCourse($course);
60
61          $lessonB = new Lesson;
62          $lessonB->setName('Datas');
63          $lessonB->setDescription('Aula sobre Datas');
64          $lessonB->setCourse($course);
65
66          $lessonCollection = new ArrayCollection;
67          $lessonCollection->add($lessonA);
68          $lessonCollection->add($lessonB);
69          $userA->setLessonCollection($lessonCollection);
70
71          $enrollment = new Enrollment;
72          $enrollment->setUser($userA);
73          $enrollment->setCourse($course);
74
75          $progressA = new Progress;
76          $progressA->setPercent(100);
77          $progressA->setUser($userA);
78          $progressA->setLesson($lessonA);
79          $progressA->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
80  Y-m-d H:i:s')));
81
82          $progressB = new Progress;
83          $progressB->setPercent(90);
84          $progressB->setUser($userA);
```

```
85          $progressB->setLesson($lessonB);
86          $progressB->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
87  Y-m-d H:i:s')));
88
89          $this->getEntityManager()->persist($userA);
90          $this->getEntityManager()->persist($enrollment);
91          $this->getEntityManager()->persist($progressA);
92          $this->getEntityManager()->persist($progressB);
93          $this->getEntityManager()->flush();
94
95          $qb = $this->em->createQueryBuilder();
96
97          $qb->select('p')
98             ->from('DoctrineNapratica\Model\Progress', 'p')
99             ->innerJoin('p.lesson', 'l')
100            ->innerJoin('l.course', 'c')
101            ->where('p.percent = 100');
102
103         $query = $qb->getQuery();
104
105         $result = $query->getResult();
106
107         $this->assertEquals(1, count($result));
108
109         $this->assertEquals($userA->getId(), $result[0]->getUser()->getId());
110         $this->assertEquals($lessonA->getId(), $result[0]->getLesson()->getId());
111     }
112
113     //testa a dql dos Users com parametros
114     public function testUserParameters()
115     {
116         $userA = $this->buildUser('userA');
117         $userB = $this->buildUser('userB');
118
119         $this->getEntityManager()->persist($userA);
120         $this->getEntityManager()->persist($userB);
121         $this->getEntityManager()->flush();
122
123         $qb = $this->em->createQueryBuilder();
124         $qb->select('u')
125            ->from('DoctrineNapratica\Model\User', 'u')
126            ->where('u.login = :login or u.email = :email')
```

```
127             ->setParameters(array('login' => 'userA', 'email' => 'userA@domain.co\
128 m'));
129 
130         $query = $qb->getQuery();
131         $users = $query->getResult();
132 
133         $this->assertEquals(1, count($users));
134 
135         $this->assertEquals($userA->getId(), $users[0]->getId());
136 
137         $this->assertInstanceOf(get_class($userA), $users[0]);
138 
139         //outra forma de escrever a query
140         $qb = $this->em->createQueryBuilder();
141         $qb->select('u')
142            ->from('DoctrineNapratica\Model\User', 'u')
143            ->where($qb->expr()->orX(
144                 $qb->expr()->eq('u.login', ':login'),
145                 $qb->expr()->eq('u.email', ':email')
146             ))
147            ->setParameters(array('login' => 'userA', 'email' => 'userA@domain.co\
148 m'));
149 
150         $query = $qb->getQuery();
151         $users = $query->getResult();
152 
153         $this->assertEquals(1, count($users));
154 
155         $this->assertEquals($userA->getId(), $users[0]->getId());
156 
157         $this->assertInstanceOf(get_class($userA), $users[0]);
158     }
159 
160     //cria um User
161     private function buildUser($login)
162     {
163         $user = new User;
164         $user->setName($login . ' name');
165         $user->setLogin($login);
166         $user->setEmail($login . '@domain.com');
167         $user->setAvatar($login . '.png');
168 
```

```
169         return $user;
170     }
171
172     //cria um Course
173     private function buildCourse($courseName)
174     {
175         $teacher = $this->buildUser();
176         //login é unique
177         $teacher->setLogin('jobs'.$courseName);
178
179         $course = new Course;
180         $course->setName($courseName);
181         $course->setDescription('Curso de PHP');
182         $course->setValue(100);
183         $course->setTeacher($teacher);
184
185         return $course;
186     }
187 }
```

https://gist.github.com/eminetto/7364021[33]

No método *testUserParameters* podemos ver uma outra forma de criação de consultas:

```
1 $qb = $this->em->createQueryBuilder();
2 $qb->select('u')
3    ->from('DoctrineNapratica\Model\User', 'u')
4    ->where($qb->expr()->orX(
5         $qb->expr()->eq('u.login', ':login'),
6         $qb->expr()->eq('u.email', ':email')
7     ))
8    ->setParameters(array('login' => 'userA', 'email' => 'userA@domain.com'));
```

O método *expr* nos permite criar cláusulas *where* avançadas. Ele fornece métodos para criarmos condicionais como *or* e *and* (*orX* e *andX*), assim como métodos de comparação como igual (*eq*), diferente (*neq*), maior que (*gt*), nulo (*isNull*) e cálculos como soma (*sum*) e multiplicação (*prod*). Trata-se de uma funcionalidade poderosa, que permite a criação de consultas ainda mais dinâmicas e flexíveis. Podemos ver mais detalhes sobre a classe *expr* na documentação oficial[34].

[33] https://gist.github.com/eminetto/7364021

[34] http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html#the-expr-class

Hydrators

Como vimos nos tópicos deste capítulo, o uso de *hydrators* é de grande importância para a performance das nossas aplicações usando o Doctrine. Com o *QueryBuilder* também podemos usar este recurso. Vamos adicionar novos testes para demonstrar o uso dos tipos de hydrators:

```
1  //testa os hydrators, limit e order
2  public function testHydrators()
3  {
4      $users = array();
5      for($i=0; $i<30; $i++) {
6          $users[$i] = $this->buildUser('user' . $i);
7          $this->getEntityManager()->persist($users[$i]);
8      }
9      $this->getEntityManager()->flush();
10
11     $qb = $this->em->createQueryBuilder();
12     //retorna os 10 primeiros ordenados por name
13     $qb->select('u')
14        ->from('DoctrineNapratica\Model\User', 'u')
15        ->setMaxResults(10)
16        ->orderBy('u.name', 'ASC');
17
18     $query = $qb->getQuery();
19     $result = $query->getResult();
20
21     $this->assertEquals(10, count($result));
22
23     //retorna os dados no formato de um array de objetos (é o padrão)
24     $result = $query->getResult();
25     $this->assertInstanceOf(get_class($users[0]), $result[0]);
26     $this->assertEquals($users[0]->getId(), $result[0]->getId());
27
28     //retorna os dados na forma de um array de arrays
29     $result = $query->getArrayResult();
30     $this->assertTrue(is_array($result[0]));
31     $this->assertEquals($users[0]->getId(), $result[0]['id']);
32
33     //O alias da entidade é adicionado ao indice do array.
34     $result = $query->getScalarResult();
35     $this->assertTrue(is_array($result[0]));
36     $this->assertEquals($users[0]->getId(), $result[0]['u_id']);
37
```

```
38      //se a consulta retorna apenas um elemento podemos usar o single_scalar
39      $qb = $this->em->createQueryBuilder();
40      $qb->select('u.login')
41         ->from('DoctrineNapratica\Model\User', 'u')
42         ->setMaxResults(1);
43      $query = $qb->getQuery();
44      $result = $query->getSingleScalarResult();
45      $this->assertEquals($users[0]->getLogin(), $result);
46  }
```

https://gist.github.com/eminetto/7364026[35]

No código acima podemos ver o uso dos métodos *getResult*, *getArrayResult*, *getScalarResult*, *getSingleScalarResult* que fazem o trabalho de retornar os resultados no formato que mais se adequa ao nosso projeto. É importante fazermos o uso adequado dos *hydrators* para garantir uma boa performance do aplicativo, conforme vamos ver no capítulo específico sobre performance deste livro.

[35] https://gist.github.com/eminetto/7364026

Eventos

Introdução

Se você já trabalhou com algum sistema gerenciador de banco de dados mais avançado como *MySQL*, *PostgreSQL*, *Oracle* ou *MSSql* já deve ter em algum momento esbarrado com o conceito de *triggers*. As *triggers* são eventos que acontecem durante o ciclo de manipulação dos dados como a criação, atualização e exclusão. Podemos interceptar eventos em cada fase deste ciclo e colocarmos lógica de programação para ser executada antes da criação de um registro (*PRE-INSERT*), ou após a atualização (*POST-INSERT*), por exemplo.

Com o Doctrine podemos fazer esta mesma funcionalidade, com a vantagem de estarmos usando objetos e nosso código funcionar em todos os sistemas gerenciadores de banco de dados, inclusive os que não possuem *triggers*.

Vamos incrementar nosso projeto adicionando o seguinte evento: quando for inserido ou atualizado uma instância de *Progress* e o atributo *percent* for igual a 100 vamos atualizar a entidade *Enrollment* e gerar o *certificationCode*. Assim automaticamente vamos gerar os certificados para nossos alunos, conforme eles terminarem de assistir nossos cursos.

Configurando os eventos

O primeiro passo necessário é alterar o *bootstrap.php* e o *tests/bootstrap.php* para habilitarmos o gerenciamento de eventos do Doctrine.

No *bootstrap.php* incluímos as seguintes linhas ao final do arquivo:

```
1  //subscriber
2  $evm = $entityManager->getEventManager();
3  $entitySubscriber = new DoctrineNaPratica\Model\Subscriber\EntitySubscriber;
4  $evm->addEventSubscriber($entitySubscriber);
```

https://gist.github.com/eminetto/7364035[36]

No *tests/bootstrap.php* alteramos as últimas linhas do arquivo:

[36] https://gist.github.com/eminetto/7364035

```php
//cria o entityManager
$entityManager = EntityManager::create($dbParams, $config);

$evm = $entityManager->getEventManager();
$entitySubscriber = new DoctrineNaPratica\Model\Subscriber\EntitySubscriber;
$evm->addEventSubscriber($entitySubscriber);

return $entityManager;
```

https://gist.github.com/eminetto/7364038[37]

O responsável pelo trabalho de gerenciamento dos eventos é a classe *EventManager*, que nos é fornecida pelo *EntityManager*. Precisamos criar uma classe que vai "ouvir" todos os eventos que as nossas entidades geram (criação, alteração, exclusão, etc) e indicar ao *EventManager* que ela "assina" estes eventos. Esta classe, que iremos criar a seguir, é a *EntitiSubscriber*, conforme o código que acabamos de incluir no nosso *bootstrap*.

Vamos agora criar a classe *EntitySubscriber*. Primeiro precisamos criar o diretório:

```
mkdir src/DoctrineNaPratica/Model/Subscriber
```

E criamos o arquivo *EntitySubscriber.php*:

```php
<?php

namespace DoctrineNaPratica\Model\Subscriber;

use Doctrine\Common\EventSubscriber;
use Doctrine\Common\EventArgs;
use Doctrine\ORM\Event;
use Doctrine\ORM\Events;

/**
 * Classe base que vai ouvir todos os eventos relacionados as entidades
 *
 */
class EntitySubscriber implements EventSubscriber
{

    /**
     * Lista de entidades a serem monitoradas
```

[37] https://gist.github.com/eminetto/7364038

```
19          * @var array
20          */
21         private $listenedEntities = array(
22             'DoctrineNaPratica\Model\Progress',
23         );
24 
25         /**
26          * Lista de eventos a serem monitorados
27          * @var  array
28          */
29         public function getSubscribedEvents()
30         {
31             return array(
32                 Events::onFlush,
33                 Events::preUpdate,
34                 Events::postUpdate,
35                 Events::postPersist
36             );
37         }
38 
39         /**
40          * Verifica se a entidade sendo alterada é uma das monitoradas
41          *
42          * @var boolean
43          */
44         protected function isListenedTo($entity)
45         {
46             $entityClass = get_class($entity);
47             if (in_array($entityClass, $this->listenedEntities)) {
48                 return true;
49             }
50 
51             return false;
52         }
53 
54 
55         /**
56          * Verifica o progresso geral do usuário no curso
57          * @param  Course $course Curso
58          * @param  User $user   Usuário
59          * @param  EntityManager $em     EntityManager
60          * @return int         Percentual de progresso
```

```
61          */
62         private function getCourseProgress($course, $user, $em)
63         {
64                 $lessons = $course->getLessonCollection();
65                 $lessonIds = array();
66                 foreach ($lessons as $l) {
67                         $lessonIds[] = $l->getId();
68                 }
69
70                 //busca o progresso do aluno no curso
71                 $qb = $em->createQueryBuilder();
72             $qb->select('SUM(p.percent) AS percent')
73                ->from('DoctrineNapratica\Model\Progress', 'p')
74                ->where($qb->expr()->andX(
75                     $qb->expr()->in('p.lesson', ':lessonIds'),
76                     $qb->expr()->eq('p.user', ':user')
77                 ))
78                ->setParameters(array('lessonIds' => $lessonIds, 'user' => $user));
79
80             $query = $qb->getQuery();
81             $percent = $query->getSingleScalarResult();
82             return $percent;
83         }
84
85         /**
86          * Método que será executado após a inclusão de uma entidade
87          * @param  EventArgs $args Argumentos do evento
88          */
89         public function postPersist(EventArgs $args)
90         {
91             //só faz o cálculo caso a entidade sendo salva é a monitorada
92             if ( ! $this->isListenedTo($args->getEntity())) return;
93
94             $e = $args->getEntity();
95             $em = $args->getEntityManager();
96             $course = $e->getLesson()->getCourse();
97             $user = $e->getUser();
98             $numberOfLessons = count($e->getLesson()->getCourse()->getLessonCollecti\
99 on());
100             $courseProgress = $this->getCourseProgress($course, $user, $em);
101             //verifica se progresso do aluno no curso é igual ao total do curso
102             if (($numberOfLessons * 100) == $courseProgress) {
```

```
103                 //pega o Enrollment do usuário
104                 $enrollment = $em->getRepository('DoctrineNaPratica\Model\Enrollment')
105                                             ->findOneBy(array(
106                                 'user' => $user, 'course' => $course)
107                             );
108                     $certificationCode = md5($user->getId() . $course->getId());
109                 $enrollment->setCertificationCode($certificationCode);
110                 $em->persist($enrollment);
111                 $em->flush();
112             }
113     }
114
115     /**
116      * Método que será executado antes da atualização das entidades
117      * @param  EventArgs $args Argumentos do evento
118      */
119     public function preUpdate(EventArgs $args)
120     {
121
122     }
123
124     /**
125      * Método que será executado após a inclusão ou alteração
126      * @param  EventArgs $args Argumentos do evento
127      */
128     public function onFlush(EventArgs $args)
129     {
130
131     }
132
133     /**
134      * Método que será executado após a atualização das entidades
135      * @param  EventArgs $args Argumentos do evento
136      */
137     public function postUpdate(EventArgs $args)
138     {
139
140     }
141 }
```

https://gist.github.com/eminetto/7364043[38]

[38] https://gist.github.com/eminetto/7364043

Quando indicamos ao *EventManager* que estamos usando a classe *EntitySubscriber* para gerenciar nossos eventos ela será invocada para qualquer evento de qualquer entidade. Como queremos gerenciar apenas os eventos da classse *Progress* é preciso definirmos isso no atributo *$listenedEntities* que será lido pelo método *isListenedTo* e vai definir se o evento deve ou não ser tratado.

No método *getSubscribedEvents* definimos quais eventos queremos observar. Neste exemplo estamos ouvindo os eventos:

- *onFlush*. Evento que ocorre no momento que os dados são confirmados na base de dados, quando o *EntityManager* executa o método *flush()*.
- *preUpdate*. Evento que acontece antes da atualização da entidade.
- *postUpdate*. Evento que acontece após a atualização da entidade.
- *postPersist*. Evento que acontece após a criação da entidade.

Para o nosso objetivo, calcular o certificado do aluno, estamos usando o método *postPersist*, onde colocamos toda a lógica do nosso cálculo. Neste nosso exemplo não seria necessário o uso dos outros eventos, então poderíamos ter removido as linhas correspondentes no método *getSubscribedEvents* para melhorar a performance do aplicativo.

Testando

Vamos agora criar um teste unitário para verificar se nossos eventos estão sendo monitorados. Criamos o arquivo *tests/src/DoctrineNaPratica/Model/EventTest.php*:

```
1  <?php
2
3  namespace DoctrineNaPratica\Model;
4
5  use DoctrineNaPratica\Test\TestCase;
6  use DoctrineNaPratica\Model\User;
7  use DoctrineNaPratica\Model\Course;
8  use DoctrineNaPratica\Model\Lesson;
9  use DoctrineNaPratica\Model\Progress;
10 use DoctrineNaPratica\Model\Enrollment;
11 use Doctrine\Common\Collections\ArrayCollection;
12
13 /**
14  * @group Model
15  */
16 class EventTest extends TestCase
17 {
```

```
18      //testa a execução do evento
19      public function testUserCertificate()
20      {
21          $userA = $this->buildUser('UserA');
22          $teacher = $this->buildUser('Teacher');
23
24          $course = new Course;
25          $course->setName('PHP');
26          $course->setDescription('Curso de PHP');
27          $course->setValue(100);
28          $course->setTeacher($teacher);
29
30          $lessonA = new Lesson;
31          $lessonA->setName('Arrays');
32          $lessonA->setDescription('Aula sobre Arrays');
33          $lessonA->setCourse($course);
34
35          $lessonB = new Lesson;
36          $lessonB->setName('Datas');
37          $lessonB->setDescription('Aula sobre Datas');
38          $lessonB->setCourse($course);
39
40          $enrollment = new Enrollment;
41          $enrollment->setUser($userA);
42          $enrollment->setCourse($course);
43
44          $lessonCollection = new ArrayCollection;
45          $lessonCollection->add($lessonA);
46          $lessonCollection->add($lessonB);
47          $userA->setLessonCollection($lessonCollection);
48          $course->setLessonCollection($lessonCollection);
49
50          $enrollmentCollection = new ArrayCollection;
51          $enrollmentCollection->add($enrollment);
52          $userA->setEnrollmentCollection($enrollmentCollection);
53
54          $progressA = new Progress;
55          $progressA->setPercent(100);
56          $progressA->setUser($userA);
57          $progressA->setLesson($lessonA);
58          $progressA->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
59  Y-m-d H:i:s')));
```

```
60
61          $this->getEntityManager()->persist($userA);
62          $this->getEntityManager()->persist($teacher);
63          $this->getEntityManager()->persist($enrollment);
64          $this->getEntityManager()->persist($progressA);
65          $this->getEntityManager()->flush();
66
67          $savedUser = $this->getEntityManager()->find(get_class($userA), 1);
68          $enrollments = $savedUser->getEnrollmentCollection();
69          //deve ser nulo porque o progresso da segunda lição não é 100
70          $this->assertNull($enrollments->first()->getCertificationCode());
71
72          $progressB = new Progress;
73          $progressB->setPercent(100);
74          $progressB->setUser($userA);
75          $progressB->setLesson($lessonB);
76          $progressB->setCreated(\DateTime::createFromFormat('Y-m-d H:i:s', date('\
77  Y-m-d H:i:s')));
78          $this->getEntityManager()->persist($progressB);
79          $this->getEntityManager()->flush();
80
81          $savedUser = $this->getEntityManager()->find(get_class($userA), 1);
82          $enrollments = $savedUser->getEnrollmentCollection();
83          $certificationCode = md5($userA->getId() . $course->getId());
84
85          //deve ser diferente de nulo
86          $this->assertEquals($certificationCode, $enrollments->first()->getCertif\
87  icationCode());
88      }
89
90      //cria um User
91      private function buildUser($login)
92      {
93          $user = new User;
94          $user->setName($login . ' name');
95          $user->setLogin($login);
96          $user->setEmail($login . '@domain.com');
97          $user->setAvatar($login . '.png');
98
99          return $user;
100     }
101 }
```

https://gist.github.com/eminetto/7364045[39]

Com o teste podemos comprovar que no momento da inclusão de um novo progresso para o aluno o evento é executado e o código do certificado é gerado.

Mas os códigos do *EntitySubscriber* e o seu teste não contemplam alguns casos como a alteração de um progresso ou mesmo a exclusão do mesmo. Nestas situações deve ser testado se o aluno ainda possui o direito ao certificado e o *certificationCode* deve ser alterado caso necessário. Deixo essa refatoração como desafio para o leitor. É preciso mudar o *EntitySubscriber* para atender a estes casos, bem como o *EventTest* para realizar a conferência do código alterado.

[39] https://gist.github.com/eminetto/7364045

Data Fixtures

O Doctrine fornece mais um recurso interessante, chamado *Data Fixtures*. Podemos usar este recurso para popular nossa base de dados com informação que pode ser usada em testes unitários ou mesmo em produção. Como exemplo podemos citar uma tabela de endereços ou cidades do Brasil, que pode ser carregada automaticamente no momento do *deploy* do projeto para a produção, ou um conjunto de usuários e cursos que pode ser usado para testar alguma lógica de programação em algum teste unitário.

Como exemplo vamos criar uma *fixture* que vai popular nossa base de dados com alguns usuários e cursos.

O recurso de *fixtures* é um pacote adicional do Doctrine, por isso precisamos incluir seu uso no nosso *composer.json*:

```
"doctrine/data-fixtures": "*"
```

E atualizar as dependências usando o *php composer.phar update*

Agora vamos criar a *fixture* para carregar os usuários. Vamos criar um novo diretório para armazenarmos as nossas classes de *fixtures*:

```
mkdir src/DoctrineNaPratica/Model/Fixture
```

O conteúdo do arquivo *src/DoctrineNaPratica/Model/Fixture/User.php* é:

```
<?php
namespace DoctrineNaPratica\Model\Fixture;

use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\FixtureInterface;
use DoctrineNaPratica\Model\User as ModelUser;

/**
 * Cria alunos para popular a base
 */
class User implements FixtureInterface
{
    private $data = array(
```

```
14          0 => array(
15              'name' => 'Elton Minetto',
16              'login' => 'eminetto',
17              'email' => 'eminetto@coderockr.com',
18          ),
19          1 => array(
20              'name' => 'Steve Jobs',
21              'login' => 'steve',
22              'email' => 'jobs@coderockr.com',
23          ),
24          2 => array(
25              'name' => 'Bill Gates',
26              'login' => 'bill',
27              'email' => 'bill@coderockr.com',
28          ),
29      );
30
31      public function load(ObjectManager $manager)
32      {
33          foreach ($this->data as $d) {
34              $user = new ModelUser;
35              $user->setName($d['name']);
36              $user->setLogin($d['login']);
37              $user->setEmail($d['email']);
38              $manager->persist($user);
39          }
40          $manager->flush();
41      }
42  }
```

https://gist.github.com/eminetto/7364055[40]

A classe é relativamente simples, sendo necessário apenas que ela implemente a interface *FixtureInterface* e o método *load*, onde deve conter a lógica que salva os dados na base de dados.

A *fixture* para carregar cursos é bem similar a de usuários. O arquivo *src/DoctrineNaPratica/Model/Fixture/Course.php* contém:

[40] https://gist.github.com/eminetto/7364055

```php
<?php
namespace DoctrineNaPratica\Model\Fixture;

use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\FixtureInterface;
use DoctrineNaPratica\Model\Course as ModelCourse;
use DoctrineNaPratica\Model\Lesson;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * Cria cursos para popular a base
 */
class Course implements FixtureInterface
{
    private $data = array(
        0 => array(
            'name' => 'Doctrine na prática',
            'value' => 666,
            'teacher' => array(
                'login' => 'eminetto',
            ),
            'lessons' => array(
                0 => array(
                    'name' => 'Introdução',
                    'description' => 'Introdução ao Doctrine',
                ),
                1 => array(
                    'name' => 'Instalação',
                    'description' => 'Instalando o Doctrine',
                ),
            )
        ),
        1 => array(
            'name' => 'Zend Framework 2 na prática',
            'value' => 666,
            'teacher' => array(
                'login' => 'eminetto',
            ),
            'lessons' => array(
                0 => array(
                    'name' => 'Introdução',
                    'description' => 'Introdução ao ZF2',
```

```
43                 ),
44                 1 => array(
45                     'name' => 'Instalação',
46                     'description' => 'Instalando o ZF2',
47                 ),
48             )
49         ),
50     );
51 
52     public function load(ObjectManager $manager)
53     {
54         foreach ($this->data as $d) {
55             $course = new ModelCourse;
56             $course->setName($d['name']);
57             $course->setValue($d['value']);
58             $teacher = $manager->getRepository('DoctrineNaPratica\Model\User')
59                                ->findOneBy(array('login' => $d['teacher']['login\
60 ']));
61             $course->setTeacher($teacher);
62             $lessons = new ArrayCollection;
63             foreach ($d['lessons'] as $l) {
64                 $lesson = new Lesson;
65                 $lesson->setName($l['name']);
66                 $lesson->setDescription($l['description']);
67                 $lesson->setCourse($course);
68                 $lessons->add($lesson);
69             }
70             $course->setLessonCollection($lessons);
71             $manager->persist($course);
72         }
73         $manager->flush();
74     }
75 }
```

https://gist.github.com/eminetto/7364062[41]

O método *load* da fixture de cursos é um pouco mais complexa pois precisa consultar a base de dados para encontrar o usuário que será o professor do curso.

Além de criarmos as classes precisamos criar um arquivo que irá instanciar e executar as *fixtures*. Para isto criamos o arquivo *fixtures.php*, que contém o código:

[41] https://gist.github.com/eminetto/7364062

```php
<?php
// Setup autoloading
include __DIR__ . '/bootstrap.php';
include __DIR__ . '/cli-config.php';

use Doctrine\Common\DataFixtures\Loader;
use Doctrine\Common\DataFixtures\Executor\ORMExecutor;
use Doctrine\Common\DataFixtures\Purger\ORMPurger;
use DoctrineNaPratica\Model\Fixture\Course as FixtureCourse;
use DoctrineNaPratica\Model\Fixture\User as FixtureUser;

$loader = new Loader();
//também podemos carregar todos os fixtures do diretório indicado
// $loader->loadFromDirectory(__DIR__ . '/src/DoctrineNaPratica/Model/Fixture');
$loader->addFixture(new FixtureUser);
$loader->addFixture(new FixtureCourse);

$purger = new ORMPurger();
$executor = new ORMExecutor($entityManager, $purger);
$executor->execute($loader->getFixtures(), true);
```

https://gist.github.com/eminetto/7364066[42]

Começamos incluindo nosso *bootstrap.php* e também o *cli-config.php* pois trata-se de uma ferramenta que será executada na linha de comando. Com estes dois *includes* fazemos as configurações necessárias de carregamento automático de classes bem como a criação de um *EntityManager* que será necessário nas próximas linhas.

Precisamos criar uma instância da classe *DoctrineCommonDataFixturesLoader* que é responsável pela carga e verificação das nossas classes de *fixtures*. Temos duas formas de dizermos ao *loader* quais classes ele deve analisar e carregar. A primeira é usando o método:

```php
$loader->loadFromDirectory(__DIR__ . '/src/DoctrineNaPratica/Model/Fixture');
```

Desta forma o *loader* vai ler o diretório e carregar todas as classes que criarmos. O único ponto a ser considerado neste caso é a ordem que as fixtures precisam ser executadas. A *fixture* de cursos deve ser executada após a de usuários para que funcione da maneira que esperamos. Se usarmos o método *loadFromDirectory* precisamos definir a ordem de execução das *fixtures* no código das classes, conforme o exemplo abaixo retirado da documentação oficial:

[42] https://gist.github.com/eminetto/7364066

```php
namespace MyDataFixtures;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;

class MyFixture extends AbstractFixture implements OrderedFixtureInterface
{
    public function load(ObjectManager $manager)
    {}

    public function getOrder()
    {
        return 10; // number in which order to load fixtures
    }
}
```

Desta forma o *loader* sabe qual a ordem a ser carregada e posteriormente executada.

A outra forma é usando o método *addFixture*, conforme o exemplo do *fixtures.php* e assim indicando explicitamente a ordem que esperamos que as *fixtures* sejam executadas.

Precisamos também criar uma instância da classe *DoctrineCommonDataFixturesPurgerORMPurger* e uma da classe *DoctrineCommonDataFixturesExecutorORMExecutor*. O *ORMPurger* é responsável por limpar a base de dados antes das *fixtures* serem executadas e o *ORMExecutor* contém a lógica responsável por fazer com que os métodos *load* das nossas classes sejam executadas.

O segundo parâmetro do comando:

```php
$executor->execute($loader->getFixtures(), true);
```

indica que vai ser realizado o *append* dos dados e não o *purge*, ou seja, os dados não vão ser excluidos das tabelas. Para que a base seja excluída antes da execução basta mudar este parâmetro para para *false* e o objeto *ORMPurger* vai fazer seu trabalho.

Agora podemos executar os fixtures com o comando:

```
php fixtures.php
```

Integrando com frameworks

Graças ao advento do *Composer* a integração do Doctrine com a maioria dos frameworks modernos é muito simples. Neste capítulo vamos ver como integrá-lo com alguns exemplares desta nova safra de frameworks.

Zend Framework 2

Vamos começar criando um pequeno projeto com o Zend Framework 2 e para isso usaremos o *ZFTool*, ferramenta que auxilia na criação de projetos. O primeiro passo é criarmos um arquivo *composer.json* requerindo a instalação do *ZFTool* dentro de um diretório específico para nosso exemplo:

```
mkdir zf2_doctrine
cd zf2_doctrine
```

Dentro deste diretório criamos o arquivo *composer.json*:

```
{
    "require": {
        "zendframework/zftool": "dev-develop"
    }
}
```

E agora podemos usar o comando abaixo para instalarmos o *ZFTool*:

```
php composer.phar install
```

Com o *ZFTool* instalado vamos criar um novo projeto de exemplo:

```
php vendor/bin/zf.php create project ZF2Doctrine
cd ZF2Doctrine/
php composer.phar install
```

Agora temos um projeto básico instalado com todas as suas dependências.

A forma mais fácil de realizar a integração do Zend Framework 2 com o Doctrine é usando-se o conceito de *Service Managers* do framework. O *Service Manager* é um registro onde podemos descrever como uma determinada classe ou porção de códigos precisa ser inicializada antes de podermos usá-la. Desta forma, iremos definir o *EntityManager* do Doctrine como um serviço que poderemos usar em qualquer lugar do framework.

Mas antes precisamos instalar o próprio Doctrine usando o *composer.json* do nosso projeto. Vamos alterar o arquivo *ZF2Doctrine/composer.json* para que ele inclua o Doctrine:

```
{
    "name": "zendframework/skeleton-application",
    "description": "Skeleton Application for ZF2",
    "license": "BSD-3-Clause",
    "keywords": [
        "framework",
        "zf2"
    ],
    "homepage": "http://framework.zend.com/",
    "require": {
        "php": ">=5.3.3",
        "zendframework/zendframework": "2.2.*",
        "doctrine/common": "2.4.*",
        "doctrine/dbal": "2.4.*",
        "doctrine/orm": "2.4.*"
    }
}
```

https://gist.github.com/eminetto/7364085[43]

E mandamos o *Composer* atualizar as dependências:

```
php composer.phar update
```

Vamos agora configurar o *ServiceManager* do Framework, alterando o arquivo *module/Application/config/module.config.php*. Este é o arquivo de configuração do módulo *Application*, o principal módulo do projeto, e é estruturado na forma de um grande *Array* de dados. Um dos itens do *Array* é o *service_manager* que iremos alterar para o seguinte:

```
'service_manager' => array(
    'abstract_factories' => array(
        'Zend\Cache\Service\StorageCacheAbstractServiceFactory',
        'Zend\Log\LoggerAbstractServiceFactory',
    ),
    'aliases' => array(
        'translator' => 'MvcTranslator',
    ),
    'factories' => array(
        'EntityManager' => function($sm) {
            $isDevMode = true;
            // configurações do banco de dados
```

[43] https://gist.github.com/eminetto/7364085

```
13              $dbParams = array(
14                  'driver'   => 'pdo_mysql',
15                  'user'     => 'root',
16                  'password' => 'root',
17                  'dbname'   => 'dnp',
18                  'host'     => 'localhost',
19                  'port'     => '3306',
20              );
21              $config = \Doctrine\ORM\Tools\Setup::createConfiguration($isDevMode);
22              $paths = array(__DIR__ . '/../src/Application/Model');
23              //leitor das annotations das entidades
24              $driver = new \Doctrine\ORM\Mapping\Driver\AnnotationDriver(
25                  new \Doctrine\Common\Annotations\AnnotationReader(),
26                  $paths
27              );
28              $config->setMetadataDriverImpl($driver);
29              //registra as annotations do Doctrine
30              \Doctrine\Common\Annotations\AnnotationRegistry::registerFile(
31                  __DIR__ . '/../../../vendor/doctrine/orm/lib/Doctrine/ORM/Mappin\
32  g/Driver/DoctrineAnnotations.php'
33              );
34              //cria o entityManager
35              $entityManager = \Doctrine\ORM\EntityManager::create($dbParams, $con\
36  fig);
37
38              return $entityManager;
39          }
40      ),
41  ),
```

https://gist.github.com/eminetto/7364090[44]

Basicamente o que fizemos foi pegar o conteúdo do arquivo *bootstrap.php* e colocá-lo dentro da configuração do *ServiceManager*. Desta forma, sempre que requisitarmos o serviço chamado *EntityManager* o próprio framework vai se responsabilizar pela execução deste trecho de código e nos entregar um *EntityManager* funcional. O *ServiceManager* do Zend Framework é inteligente o suficiente para identificar se já foi criada uma instância do serviço e nos entregá-la caso exista, não sendo necessária a execução deste trecho de código cada vez que o requisitarmos, o que aumenta a performance da aplicação.

Podemos agora simplesmente copiar as nossas entidades criadas nos capítulos anteriores para dentro do módulo *Application*, mais especificamente no diretório *src/Application/Model*, conforme nós indicamos na variável *$paths* dentro da configuração do serviço *EntityManager*.

[44] https://gist.github.com/eminetto/7364090

Com isso podemos basta usarmos o *Service Manager* para nos entregar o *EntityManager* e usarmos o Doctrine como fizemos anteriormente. Em um *controller*, por exemplo, podemos fazer desta forma:

```
1  <?php
2
3  namespace Application\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class IndexController extends AbstractActionController
9  {
10     public function indexAction()
11     {
12             $em = $this->getServiceLocator()->get('EntityManager');
13             $user = $em->find('Application\Model\User', 1);
14
15         return new ViewModel(array('user' => $user));
16     }
17 }
```

https://gist.github.com/eminetto/7364102[45]

E nossa entidade *User* pode ser usada pela *view* do Zend Framework para imprimir os dados do usuário, como no exemplo da *index.phtml*:

```
1  <?php echo $user->getName();?>
```

Silex

O Silex é um micro-framework construido com pedaços de outro projeto maior, o Symfony 2. Por ser um micro-framework ele concentra-se em realizar as coisas mais básicas de um projeto Web, principalmente as relacionadas com roteamento e requisições. O restante das funcionalidades, como modelos, visões, log, etc, podem ser atendidas acoplando-se outros componentes, como o próprio Doctrine.

Como todo projeto moderno, o Silex também faz uso do Composer para realizar sua instalação e integração com outros pacotes, então vamos criar um diretório para nosso exemplo:

[45] https://gist.github.com/eminetto/7364102

```
mkdir silex_doctrine
```

e um composer.json para ele:

```
{
    "require": {
        "silex/silex": "1.0.*@dev",
        "doctrine/common": "2.4.*",
        "doctrine/dbal": "2.4.*",
        "doctrine/orm": "2.4.*"
    }
}
```

https://gist.github.com/eminetto/7364105[46]

O Silex possui nativamente uma integração com o componente *DBAL* do Doctrine mas não existe uma integração oficial com o *ORM*. Para suprir essa necessidade existe o projeto Open Source *Doctrine ORM Service Provider* e para usá-lo vamos adicionar mais uma linha na lista de requisitos do projeto no nosso *composer.json*:

```
"dflydev/doctrine-orm-service-provider": "1.0.*@dev",
```

Após executar o comando *php composer.phar install* podemos iniciar a configuração do projeto com o Silex.

Podemos simplesmente copiar os arquivos *bootstrap.php*, *cli-config.php* e o diretório *src* do nosso projeto inicial para usarmos no nosso projeto com o Silex, sem precisar alterar nenhuma linha.

Vamos criar nossa aplicação, criando o arquivo *index.php*:

```
<?php

use Doctrine\Common\Cache\ApcCache as Cache;
use Doctrine\Common\Collections\ArrayCollection;
use Silex\Application;
use Silex\Provider\DoctrineServiceProvider;
use Dflydev\Silex\Provider\DoctrineOrm\DoctrineOrmServiceProvider;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\HttpKernelInterface;

```

[46] https://gist.github.com/eminetto/7364105

```php
require_once __DIR__.'/bootstrap.php';

$app = new Application();

//configuração do DBAL
$app->register(new DoctrineServiceProvider, array(
    'db.options' => array(
        'driver' => 'pdo_mysql',
        'host' => '127.0.0.1',
        'port' => '3306',
        'user' => 'root',
        'password' => 'root',
        'dbname' => 'dnp'
    )
));

//configuração do ORM
$app->register(new DoctrineOrmServiceProvider(), array(
    'orm.proxies_dir' => '/tmp',
    'orm.em.options' => array(
        'mappings' => array(
            array(
                'type' => 'annotation',
                'use_simple_annotation_reader' => false,
                'namespace' => 'DoctrineNaPratica\Model',
                'path' => __DIR__ . '/src'
            )
        )
    ),
    'orm.proxies_namespace' => 'EntityProxy',
    'orm.auto_generate_proxies' => true,
    'orm.default_cache' => 'apc'
));

//rotas
$app->get('/exemplo-dbal/{id}', function ($id) use ($app) {
    $sql = "SELECT * from User WHERE id = ?";
    $user = $app['db']->fetchAssoc($sql, array((int) $id));

    return  "<h1>{$user['name']}</h1>".
            "<p>{$user['login']}</p>";
});
```

```
54
55 $app->get('/exemplo-orm/{id}', function ($id) use ($app) {
56     $em = $app['orm.em'];
57     $user = $em->find('DoctrineNaPratica\Model\User', $id);
58
59     return  "<h1>{$user->getName()}</h1>".
60             "<p>{$user->getLogin()}</p>";
61 });
62
63 $app['debug'] = true;
64 $app->run();
```

https://gist.github.com/eminetto/7364109[47]

Da linha 17 a 26 nós estamos fazendo a configuração do *DBAL* e da linha 29 a 44 configuramos o *ORM*. As configurações são parecidas com as que vimos nos outros capítulos do livro.

No restante do arquivo criamos um exemplo de uso do *DBAL* e um do *ORM*.

Se você estiver usando o PHP 5.4 ou superior pode usar o servidor embutido:

```
1 php -S 0.0.0.0:8080 index.php
```

E acessar as *URLs* no navegador: *http://localhost:8080/exemplo-dbal/1* e *http://localhost:8080/exemplo-orm/1*, mudando a última parte do endereço para mostrar um usuário que exista na sua base de dados.

O Silex é uma ótima ferramenta para criar pequenos sites ou mesmo *APIs* que serão usadas por outros projetos e a integração com o Doctrine facilita muito o processo.

[47] https://gist.github.com/eminetto/7364109

Performance

Performance é uma das preocupações que deve permear todo o desenvolvimento de qualquer projeto web, seja ele pequeno ou grande. Mesmo vivendo em uma realidade onde adicionar novos recursos computacionais tornou-se tão simples quando alguns cliques em um painel de administração, graças ao advento do *cloud computing*, precisamos pensar em formas de economizar estes recursos.

No decorrer dos capítulos deste livro vimos as vantagens de se usar um *ORM* como o Doctrine. Mas estas vantagens acabam acarretando em perda de performance, principalmente quando comparadas a sistemas que fazem acesso direto a base de dados usando *PDO*.

Neste capítulo vamos ver algumas dicas que podem melhorar bastante a performance de sua aplicação, no que diz respeito ao Doctrine.

Cache

No pacote *Common* do Doctrine existem classes que nos fornecem o acesso às principais implementações de cache, como *APC*, *Memcache*, *Redis* e *Xcache*, além da opção de usarmos o cache em arrays PHP com a classe *ArrayCache*. Usando uma destas implementações podemos melhorar a performance do Doctrine, com excessão do *ArrayCache* que serve apenas para usarmos em ambiente de testes e desenvolvimento.

Podemos usar o cache Doctrine para melhorar a performance do *ORM* de três formas: cache de consultas (*query cache*), cache de resultados (*result cache*) e cache de informações das entidades (*metadata cache*).

Query Cache

Sempre que usamos uma consulta *DQL* ela é, em algum momento, transformada em uma consulta *SQL* para ser executada no banco de dados. Com o *query cache* o Doctrine consegue armazenar as consultas *DQL* e sua correspondente *SQL* em cache, o que dispensa a conversão a cada vez que formos usá-la. Para fazermos melhor uso do *query cache* é importante criarmos consultas *DQL* usando parâmetros. Por exemplo, a query:

```
1  $qb->select('u')
2     ->from('DoctrineNapratica\Model\User', 'u')
3     ->where('u.login = :login or u.email = :email')
4     ->setParameters(array('login' => 'userA', 'email' => 'userA@domain.com'));
```

é muito melhor para fins de cache do que a query:

```
$qb->select('u')
   ->from('DoctrineNapratica\Model\User', 'u')
   ->where("u.login = 'userA' or u.email = 'userA@domain.com'");
```

A segunda consulta não é facilmente reutilizada pelo cache pois ela tornou-se muito específica e só será reutilizada caso estivermos pesquisando pelo 'userA' mais de uma vez. Enquanto que na primeira consulta o cache consegue substituir as variáveis pelo seu conteúdo em tempo de execução, o que facilita seu reuso.

Para usarmos o *query cache* basta indicarmos seu uso no nosso *bootstrap.php*, usando o método *setQueryCacheImpl* do objeto *$config*:

```
$config->setQueryCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

No exemplo estamos usando o cache armazenado em *APC* mas podemos usar uma das outras opções citadas nos parágrafos anteriores, como o *Memcache*.

Result Cache

Com o *result cache* podemos armazenar o resultado de nossas consultas para poupar o banco de dados de acessos desnecessários. Um exemplo clássico seria a lista de cidades de um estado. Quando o primeiro usuário escolher um estado podemos fazer a busca das entidades correspondentes e armazená-las no cache, assim os próximos usuários teriam a resposta pronta, diminuindo o tempo de processamento e o consumo de recursos do banco de dados.

Precisamos configurar no *bootstrap.php* o uso do *result cache*, bem como indicar em qual implementação de cache queremos que as informações sejam salvas:

```
$config->setResultCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Após configurado podemos usar o cache em nossas consultas, conforme o exemplo abaixo:

```
$query = $em->createQuery('select u from \DoctrineNaPratica\Model\User u');
$query->useResultCache(true);
```

E o Doctrine se responsabiliza por gerenciar o cache para o resultado desta consulta.

É possível configurar apenas uma query para usar o cache, sem a necessidade de mudar o *bootstrap*:

```
$query = $em->createQuery('select u from \DoctrineNaPratica\Model\User u');
$query->setResultCacheDriver(new \Doctrine\Common\Cache\ApcCache());
```

Outra opção útil é podermos indicar o tempo de vida do cache:

```
1 $query = $em->createQuery('select u from \DoctrineNaPratica\Model\User u');
2 $query->useResultCache(true);
3 $query->setResultCacheLifetime(3600);
```

Desta forma o Doctrine vai gerenciar o cache e quando os 3600 segundos terminarem ele vai considerar o resultado como expirado, realizar uma nova consulta e armazenar novamente por novos 3600 segundos.

Metadata Cache

Sempre que usamos o *EntityManager* para manipular alguma de nossas entidades ele precisa analisar o código da classe para entender a estrutura que definimos na sua criação, seja na forma de *annotations*, *YML* ou *XML*. Ao usarmos o *metadata cache* o Doctrine consegue armazenar essa informação no cache, dispensando a análise da classe toda vez que requisitarmos uma entidade, diminuindo o consumo de processamento e memória do servidor.

Para fazermos uso do *metadata cache* basta usarmos o método *setMetadataCacheImpl* do objeto *$config* no *bootstrap.php*:

```
1 $config->setMetadataCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Gerenciando o cache

Podemos usar a ferramenta de linha de comando do Doctrine para gerenciarmos o nosso cache:

Limpando todo o cache

```
1 ./vendor/bin/doctrine clear-cache
```

Limpando apenas o query cache

```
1 ./vendor/bin/doctrine clear-cache --query
```

Limpando o cache de metadados

```
1 ./vendor/bin/doctrine clear-cache --metadata
```

Limpando o cache de resultados

```
./vendor/bin/doctrine clear-cache --result
```

Estas opções são úteis em ambientes de testes e desenvolvimento mas também podem ser usadas em scripts que fazem o *deploy* da aplicação para os servidores de produção.

Cache além do ORM

Também podemos usar o cache do Doctrine para armazenarmos outras informações, além das usadas pelo *ORM*. Exemplos:

```
$cacheDriver = new \Doctrine\Common\Cache\ApcCache();
//salvando dados no cache por 3600 segundos
$cacheDriver->save('cache_id', 'my_data', 3600);
//verificando se um item está no cache
if ($cacheDriver->contains('cache_id')) {
    echo 'encontrou no cache';
} else {
    echo 'não existe no cache';
}
//recuperando dados do cache
$data = $cacheDriver->fetch('cache_id');
//excluindo dados do cache
$cacheDriver->delete('cache_id');
//limpando todo o cache
$deleted = $cacheDriver->deleteAll();
```

Assim podemos usar o cache do Doctrine para armazenar dados diversos, desde arrays, objetos, variáveis simples, conteúdo de arquivos, etc.

Definição e manipulação de entidades

Uma das fases mais importantes de qualquer projeto usando o Doctrine é a criação das entidades. E estas definições também podem fazer diferença no quesito performance.

Uma das decisões diz respeito a forma como definimos as relações entre as entidades, principalmente na opção *fetch* que vimos no capítilo **Definindo o projeto**. Neste capítulo vimos que o método padrão é o *LAZY* que faz a busca dos dados da entidade relacionada no momento que usamos o método correspondente, o *getEnrollmentCollection*, por exemplo. Ao mudarmos o parâmetro para *EXTRA_LAZY* podemos ter acesso a alguns dados das entidades relacionadas sem a necessidade de trazermos todas as entidades para a memória. Ao chamarmos o método *getEnrollmentCollection* da classe *User*, por exemplo, caso a relação for configurada como *EXTRA_LAZY* podemos usar os métodos abaixo para acessarmos apenas porções dos dados:

```
1 getEnrollmentCollection()->contains($entity)
2 getEnrollmentCollection()->count()
3 getEnrollmentCollection()->add($entity)
4 getEnrollmentCollection()->offsetSet($key, $entity)
5 getEnrollmentCollection()->slice($offset, $length = null)
```

Desta forma podemos verificar se determinada *Enrollment* está contida no conjunto (linha 1), podemos saber o número de ocorrências (linha 2), podemos adicionar uma nova entidade (linha 3 ou linha 4) ou mesmo criar um paginador dos resultados (linha 5). Tudo isso sem a necessidade de todos os registros serem buscados do banco de dados e armazenados na memória.

Outra definição que podemos fazer é definir que algumas entidades serão usadas apenas para leitura e não alteração. Para isso alteramos a definição da entidade colocando:

```
1 @ORM\Entity(readOnly=true)
```

Desta forma o *EntityManager* vai ignorar a entidade no momento de calcular as alterações e decidir se precisa fazer um update no banco de dados, mesmo se o programador alterar alguma informação de um dos objetos desta classe. Ao definirmos uma entidade como *readOnly* ainda podemos criar novos objetos e removê-los do banco de dados, mas não poderemos alterá-los. Em um projeto com muitas entidades esta definição pode poupar alguns segundos no momento do processamento das alterações realizadas durante o uso do aplicativo.

Outra dica importante é fazer uso dos *hydrators*, conforme comentamos no capítulo **Recuperação e manipulação de dados**. Recuperarmos o resultado de uma consulta na forma de arrays e não objetos pode fazer uma grande diferença no consumo de memória, principalmente quando o resultado envolve um grande conjunto de dados.

No capítulo **Eventos** vimos como podemos automatizar algumas operações usando o *EventManager* do Doctrine e como isto pode ser bem útil em diversos casos. Mas esta funcionalidade deve ser usada com cuidado pois ao inserirmos eventos no aplicativo estamos adicionado mais uma camada de lógica a ser executada a todo momento, o que pode aumentar o tempo de processamento, principalmente em aplicativos com um grande índice de modificações nas entidades.

A performance de um aplicativo é influenciada por diversos fatores, principalmente em aplicativos web que são formados por diversas camadas tais como servidores, bancos de dados, *DNS*, frontend, etc. É preciso analisar todos estes fatores que podem influenciar, mas seguindo algumas destas dicas podemos melhorar a performance da camada de acesso aos dados, onde o Doctrine tem grande influência.