ZEND FRAMEWORK 2
na prática

Elton Luís Minetto

Zend Framework 2 na prática

Elton Minetto

This book is for sale at http://leanpub.com/zend-framework2-na-pratica

This version was published on 2012-11-14

This is a Leanpub book. Leanpub helps authors to self-publish in-progress ebooks. We call this idea Lean Publishing.

To learn more about Lean Publishing, go to http://leanpub.com/manifesto.

To learn more about Leanpub, go to http://leanpub.com.

Conteúdo

Introdução

Deixe-me começar este livro explicando como funciona minha mente e minha forma de aprender. Sempre me considerei uma pessoa pragmática e isso se reflete na forma como eu aprendo as coisas. Quando me empolgo ou preciso aprender uma nova tecnologia ou ferramenta eu coloco na minha mente uma meta. Com o Zend Framework 2 não foi diferente, apesar dos motivos terem sido. A meta foi a coincidência de estar iniciando um grande projeto para um cliente exatamente na semana que o framework teve sua versão estável lançada. Com um prazo de entrega já definido pelo contrato deu-se início o desafio de aprender novos conceitos e uma nova forma de trabalhar com o código.

Tendo a meta definida eu inicio a codificação e os testes da nova tecnologia. Eu não começo lendo toda a teoria antes de colocar a "mão na massa". Eu preciso desse feedback imediato, de codificar algo pequeno rapidamente, de ter algo funcionando que me dê a vontade de continuar aprendendo. Conforme eu vou me deparando com os desafios do desenvolvimento eu paro e aí sim leio a teoria necessária para entender exatamente o que estou fazendo.

Pode não ser a melhor forma de aprender mas tem funcionado bem comigo, e baseando-se nos feedbacks da versão anterior deste e-book, parece funcionar para mais pessoas. Então vou continuar com essa abordagem neste livro. Vamos traçar uma meta inicial fácil de ser cumprida, iniciar o projeto e ir aprofundando a teoria conforme formos nos deparando com os desafios de entregar o projeto. Por isso você não vai encontrar no índice um capítulo inicial sobre teoria, explicando coisas legais como injeção de dependências ou eventos, mas vai encontrar tópicos sobre isso dentro dos capítulos sobre a codificação, conforme formos precisando usá-las.

Com essa abordagem mais prática espero levá-los pelas fases de planejamento, desenvolvimento de testes, codificação e deploy de um aplicativo web com o Zend Framework 2 e outras tecnologias úteis ao dia a dia.

Que o desafio comece!

Agradecimentos

Quero agradecer a algumas pessoas em especial.

A toda a equipe da Coderockr por me ajudar com idéias e aguentar minhas reclamações quando não encontrava algo na documentação do Zend Framework.

A equipe da Unochapecó, que acabou fornecendo a experiência de desenvolvimento de um grande projeto usando essa nova tecnologia. Em especial o Cristian Oliveira, Cledir Scopel e Lissandro Hoffmeister pelo apoio.

E minha namorada Mirian Giseli Aguiar pela revisão de português e por respeitar minha ausência e mau humor por algumas semanas.

Obrigado a todos.

Instalando

O processo de instalação do Zend Framework 2 foi um dos tópicos que teve maior avanço desde a versão anterior. Ficou realmente muito mais fácil de instalar e atualizar o framework e suas dependências.

Requisitos

Para criarmos um projeto usando o Zend Framework 2 precisamos atender os seguintes requisitos:

- Um servidor Web. O mais usado é o *Apache* mas pode ser configurado usando outros como o *IIS*. Os exemplos desse livro consideram o uso do servidor *Apache*. O *PHP 5.4* possui um servidor web embutido, mas não considerei o uso dele nesse livro pois nem todos os ambientes de desenvolvimento estão atualizados para esta versão recente da linguagem. No caso de usar o *Apache* é necessário que o módulo *mod_rewrite* esteja habilitado. No arquivo de configuração basta adicionar as linhas abaixo, ou alterá-las para refletir o seguinte:

```
1    LoadModule rewrite_module modules/mod_rewrite.so
2    AddModule mod_rewrite.c
3    AllowOverride all
```

- Um banco de dados. Não é algo obrigatório mas no nosso caso iremos usar o banco *MySQL*. Claro que você pode usar outro banco como o *SQLite* ou o *PostgreSQL*, mas os exemplos serão escritos para o *MySQL*.
- *PHP 5.3.3* ou superior.
- Extensão *intl* do *PHP*. O framework usa esta extensão para formatar datas e números. Esta extensão pode ser instalada usando-se o comando *pecl* do *PHP*.

Caso esteja usando *Windows* ou *MacOSX* estes requisitos podem ser facilmente cumpridos instalando-se um dos pacotes de desenvolvimento famosos como o*XAMPP* (*Windows* e *MacOSX*) ou o *MAMP* (MacOSX), que possuem todos os pacotes já configurados.

Usando-se *Linux* basta usar o sistema de gerenciamento de pacotes (*apt-get*, *yum*, etc) para instalar os pacotes necessários.

Instalando o framework

A forma mais recomendada de iniciar um projeto é usar um dos "esqueletos de aplicação" que estão disponíveis no *Github*. A documentação oficial do framework recomenda o uso do:

https://github.com/zendframework/ZendSkeletonApplication

O que vamos fazer nesse curso é usar um esqueleto que criei, baseado no oficial da Zend, mas com algumas novas classes que facilitam o desenvolvimento. Além disso, o esqueleto que iremos usar já vem com as configurações necessárias para usarmos testes automatizados e um módulo de modelo, com suas configurações. Venho usando esse esqueleto em aplicações reais e pretendo manter o código *open source* e atualizado.

Para iniciarmos o nosso projeto vamos clonar o projeto usando o *git*. O primeiro passo é acessarmos nosso diretório de projetos. No meu *MacOSX* esse diretório é o */Users/eminetto/Documents/Projects/* mas você pode mudá-lo para qualquer diretório do seu sistema operacional.

Vamos executar os comandos:

```
cd /Users/eminetto/Documents/Projects/
git clone git@github.com:eminetto/ZendSkeletonApplication.git zf2napratica
```

Isso vai criar um diretório chamado *zf2napratica* com o código do esqueleto.

Se você não tiver o *git* instalado na sua máquina pode fazer o download e descompactar no diretório. O download pode ser feito na url:

https://github.com/eminetto/ZendSkeletonApplication/zipball/master

Instalar dependências com Composer

Ao clonar (ou fazer o download) do esqueleto da aplicação ainda não temos o framework em si. A forma mais rápida de termos o framework instalado é usando a ferramenta *Composer*. O *Composer* é uma ferramenta criada para instalar e atualizar dependências de código em projetos *PHP*. Para entender em detalhes como funciona o *Composer* eu recomendo esse screencast e o site oficial da ferramenta.

Vamos usar o composer para instalar o framework:

```
cd zf2napratica
curl - http://getcomposer.org/installer | php
php composer.phar install
```

Com isso o *Composer* fará o download do framework e todas as suas dependências, bem como configurar um *autoloader* que o framework usará.

Bem mais fácil e rápido do que as versões antigas, que pediam cadastro no site da Zend.

Configurar o Vhosts do Apache

Um hábito que eu tenho sempre que desenvolvo um novo projeto é criar um servidor virtual na minha máquina para isolar o ambiente do projeto. Isso facilita bastante os testes, a organização dos projetos e até mesmo o *deploy* do aplicativo para o servidor de produção no final do desenvolvimento.

Para isso vamos configurar um servidor virtual no *Apache*. No arquivo *httpd.conf* (ou *apache.conf* ou na configuração de servidores virtuais do seu sistema operacional) adicionar o seguinte:

```
<VirtualHost *:80>
   ServerName zf2napratica.dev
   DocumentRoot /Users/eminetto/Documents/Projects/zf2napratica/public
   SetEnv APPLICATION_ENV "development"
   SetEnv PROJECT_ROOT "/Users/eminetto/Documents/Projects/zf2napratica"
   <Directory /Users/eminetto/Documents/Projects/zf2napratica/public>
       DirectoryIndex index.php
       AllowOverride All
       Order allow,deny
       Allow from all
   </Directory>
</VirtualHost>
```

https://gist.github.com/4003621

É necessário alterar os caminhos nas opções *DocumentRoot*, *PROJECT_ROOT* e *Directory* para refletirem o caminho correto em sua máquina.

É preciso também alterar o arquivo *hosts* do sistema operacional para adicionar o endereço *zf2napratica.dev* pois o mesmo não existe em nenhum *DNS*.

No *Linux* e *Mac OSX*, alterar o */etc/hosts* e adicionar a linha:

```
1 127.0.0.1        zf2napratica.dev
```

No Windows o arquivo que deve ser alterado é o *c:\windows\system32\drivers\etc\hosts* e a linha a ser adicionada é igual a citada acima.

Instalando o PHPUnit/PHPQATools

Vamos usar o framework *PHPUnit* para gerar testes automatizados durante o desenvolvimento. Para instalá-lo precisamos usar o *pear*, gerenciador de pacotes do *PHP*:

```
1 pear config-set auto_discover 1
2 pear install pear.phpqatools.org/phpqatools
```

Se estiver usando *Linux* ou *MacOSX* é necessário adicionar o comando *sudo* no início de cada comando acima.

Com esses passos temos um ambiente instalado e podemos iniciar o planejamento do nosso primeiro projeto usando o Zend Framework 2.

Definindo o projeto

Descrição

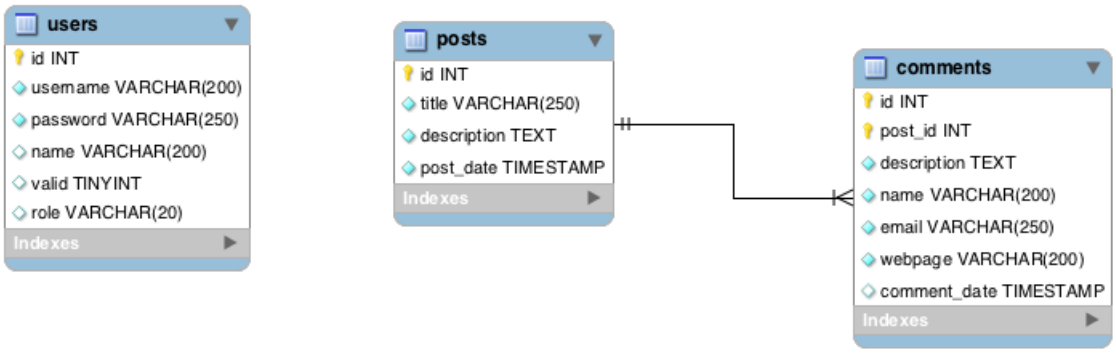
Na minha opinião a melhor forma de aprender uma nova ferramenta, linguagem ou sistema operacional é quando você realmente precisa resolver algum problema com ela. Pensando nisso, esse livro é baseado na construção de um aplicativo: um blog.

Mas um blog? Por alguns motivos:

- É um problema fácil de se entender. Todo mundo sabe como um blog funciona, seus requisitos e funcionalidades, então a fase de requisitos do projeto é fácil de completar.
- Um blog apresenta um grande número de funcionalidades comuns a vários outros sites, como módulos, controle de acesso e permissões, upload de arquivos, tratamento de formulários, cache, traduções, integração com serviços externos, etc.
- A grande maioria dos frameworks possui um exemplo "como desenvolver um blog usando X", então fica mais fácil para comparação se você já estudou algum outro framework como *CakePHP*, *CodeIgniter* ou mesmo *Ruby on Rails*

Modelagem

Agora que o convenci (ou não) que desenvolver um blog pode lhe ajudar a entender o Zend Framework, vamos mostrar a modelagem das tabelas:



Simples, como deveria ser. ## Criação das tabelas Usando alguma ferramenta, como o *PHPMyAdmin*, *SequelPro*, ou o bom e velho terminal, é possível criar a estrutura do banco usando os comandos *SQL* abaixo:

```
create database zf2napratica;
create database zf2napratica_test;

GRANT ALL privileges ON zf2napratica.* TO zend@localhost IDENTIFIED BY 'zen\
d';
GRANT ALL privileges ON zf2napratica_test.* TO zend@localhost IDENTIFIED BY\
 'zend';

use zf2napratica;

CREATE  TABLE IF NOT EXISTS `users` (
  `id` INT NOT NULL AUTO_INCREMENT ,
  `username` VARCHAR(200) NOT NULL ,
  `password` VARCHAR(250) NOT NULL ,
  `name` VARCHAR(200) NULL ,
  `valid` TINYINT NULL ,
  `role` VARCHAR(20) NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB;

CREATE  TABLE IF NOT EXISTS `posts` (
  `id` INT NOT NULL AUTO_INCREMENT ,
  `title` VARCHAR(250) NOT NULL ,
  `description` TEXT NOT NULL ,
  `post_date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB;

CREATE  TABLE IF NOT EXISTS `comments` (
  `id` INT NOT NULL AUTO_INCREMENT ,
  `post_id` INT NOT NULL ,
  `description` TEXT NOT NULL ,
  `name` VARCHAR(200) NOT NULL ,
 `email` VARCHAR(250) NOT NULL ,
 `webpage` VARCHAR(200) NOT NULL ,
  `comment_date` TIMESTAMP NULL ,
  PRIMARY KEY (`id`, `post_id`) ,
  INDEX `fk_comments_posts` (`post_id` ASC) ,
  CONSTRAINT `fk_comments_posts`
    FOREIGN KEY (`post_id` )
        REFERENCES `posts` (`id` )
      ON DELETE NO ACTION
     ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

https://gist.github.com/4011976

No script acima criamos duas bases de dados (*zf2napratica* e *zf2napratica_test*) que vamos usar para o banco de produção e o banco de testes, respectivamente. Voltaremos a esse banco de testes nos próximos tópicos.

Configurando o projeto

O *Zend Framework 2* conta com arquivos de configuração separados que são unificados no momento da execução.

Os principais arquivos que iremos usar durante o projeto são:

- *config/application.config.php*: Arquivo com as configurações gerais da aplicação. São configurações usadas por todos os módulos e componentes.
- *config/test.config.php*: Arquivo com as configurações usadas pelos testes automatizados que criaremos no decorrer do projeto.
- *config/autoload/global.php* e *config/autoload/local.php*: O arquivo *global.php* é usado como auxiliar ao *application.config.php* pois também contém configurações para a aplicação como um todo. A idéia é colocar neste arquivo configurações que podem mudar de acordo com a máquina do desenvolvedor. Um exemplo são as configurações da conexão com o banco de dados. Estas configurações podem ser alteradas para as máquinas locais, dos desenvolvedores. Para isso o desenvolvedor sobrescreve as configurações no *local.php*. O arquivo *local.php* não deve ser salvo no controle de versões (*svn* ou *git* por exemplo).
- *module/Nome/config/module.config.php*: Configurações específicas ao módulo.

Os arquivos de configuração são geralmente scripts *PHP* que retornam *arrays* de configuração. São rápidos durante a execução e de fácil leitura.

global.php

```
1  <?php
2  return array(
3      'service_manager' => array(
4          'factories' => array(
5              'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\AdapterServiceFac\
6  tory',
7      ),
8      ),
9  'db' => array(
10      'driver'         => 'Pdo',
11     'dsn'            => 'mysql:dbname=zf2napratica;host=localhost',
12     'driver_options' => array(
13         PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
14     ),
15 ));
```

https://gist.github.com/4011979

local.php

```php
<?php
    return array(
            'db' => array(
                'username' => 'zend',
                'password' => 'zend',
        )
);
```

https://gist.github.com/4011983

test.config.php

```php
<?php
return array(
    'db' => array(
        'driver' => 'PDO',
        'dsn'    => 'mysql:dbname=zf2napratica_test;host=localhost',
        'username' => 'zend',
        'password' => 'zend',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
));
```

https://gist.github.com/4011987

Voltaremos a ver esses arquivos de configuração no decorrer do projeto, e seus itens passarão a fazer mais sentido conforme formos aprendendo algumas funcionalidades do framework.

Configurações dos testes

Testes automatizados salvam sua vida!

Parece um pouco de exagero, mas o uso de testes aumenta consideravelmente a qualidade de seus códigos e garantem uma tranquilidade maior em tarefas como refatoração e melhorias. Neste projeto usaremos alguns conceitos de *TDD (Test Driven Development* (desenvolvimento guiado por testes) e usaremos a ferramenta *PHPUnit* para nos auxiliar na criação dos testes.

Novamente, o objetivo deste livro é ser prático, então não vou entrar em todos os detalhes do *TDD* e do *PHPUnit* aqui, deixando isso para excelentes livros existentes. Um link interessante para iniciar é o manual oficial do PHPUnit.

Diretório de testes

Precisamos criar o diretório onde salvaremos nossos códigos de teste. No *Linux/Mac*:

```
mkdir -p module/Application/tests/src/Application
```

Nos próximos capítulos adicionaremos os testes para nossos modelos, controladores e serviços.

Configurações do PHPUnit

Vamos usar o módulo *Skel* como modelo para os nossos novos módulos. Iniciamos usando as configurações do *PHPUnit* copiando alguns arquivos para o módulo *Application*:

```
1 cp module/Skel/tests/Bootstrap.php module/Application/tests/
2 cp module/Skel/tests/phpunit.xml module/Application/tests/
```

Precisamos agora alterar os arquivos para usarmos no novo módulo. No *phpunit.xml*, linha 8:

```
1 <testsuite name="Application Test Suite">
```

e linha 24:

```
1 <log type="coverage-html" target="_reports/coverage" title="Modulo Applicat\
2 ion" charset="UTF-8" yui="true" highlight="true" lowUpperBound="35" highLow\
3 erBound="70"/>
```

Vamos também alterar a configuração do arquivo *Bootstrap.php*:

```
1 static function getModulePath()
2 {
3     //mudar o caminho do modulo
4     return __DIR__ . '/../../../module/Application';
5 }
```

Projeto definido e configurado. Agora, mãos a obra!

Modelos

Vamos começão o desenvolvimento pela primeira camada da nossa aplicação *MVC*: os modelos. Para isso vamos criar um arquivo para cada tabela e estes devem ser armazenados no diretório *Model* do *src* do módulo, como no exemplo: *module/Application/src/Application/Model* . Todos os modelos são classes PHP que extendem a classe *Core\Model\Entity*.

Mas antes de iniciarmos criando as entidades vamos criar os testes para elas, seguindo a filosofia do *TDD*. Primeiro precisamos criar um diretório para os nossos primeiros testes. No *Linux/Mac* podemos usar o comando:

```
1  mkdir module/Application/tests/src/Application/Model
```

O diretório *tests/src* emula a estrutura do diretório *src* do módulo, assim teremos diretórios com os mesmos nomes, indicando ao que os testes se referem.

No capítulo anterior nós criamos duas bases de dados no banco *MySQL*: a *zf2napratica* e a *zf2napratica_test*. Vamos agora usar a base de dados de teste. Antes de cada teste ser executado precisamos preparar o estado inicial do banco de dados. Para isso precisamos criar um arquivo chamado *module/Application/data/test.data.php*, com o conteúdo abaixo.

Primeiro criamos o diretório *data*:

```
1  mkdir module/Application/data
```

E incluímos seu conteúdo:

```
1  <?php
2  //queries used by tests
3  return array(
4      'posts' => array(
5                      'create' => 'CREATE  TABLE if not exists posts (
6                    id INT NOT NULL AUTO_INCREMENT ,
7                    title VARCHAR(250) NOT NULL ,
8                    description TEXT NOT NULL ,
9                    post_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
10                   PRIMARY KEY (id) )
11                 ENGINE = InnoDB;',
12             'drop' => "DROP TABLE posts;"
13         ),
14 'comments' => array(
15             'create' => 'CREATE  TABLE if not exists comments (
16                   id INT NOT NULL AUTO_INCREMENT ,
17                   post_id INT NOT NULL ,
18                   description TEXT NOT NULL ,
19                   name VARCHAR(200) NOT NULL ,
20                    email VARCHAR(250) NOT NULL ,
21                   webpage VARCHAR(200) NOT NULL ,
22                   comment_date TIMESTAMP NULL ,
```

```
                    PRIMARY KEY (id, post_id) ,
                    INDEX fk_comments_posts (post_id ASC) ,
                  CONSTRAINT fk_comments_posts
                    FOREIGN KEY (post_id )
                        REFERENCES posts (id )
                        ON DELETE NO ACTION
                       ON UPDATE NO ACTION)
                ENGINE = InnoDB;',

        'drop' =>'drop table comments;'
                ),
        );
```

https://gist.github.com/4011989

Antes de cada teste ser executado as classes do módulo *Core* vão usar esse arquivo como parâmetro e criar as tabelas (usando o comando *SQL* no *create*). Após cada teste o *drop* será executado e a tabela de teste deixa de existir.

OBS: Para um projeto maior, com mais tabelas e mais testes esse procedimento pode começar a ficar demorado e complexo de manter. Para estes casos podemos usar outras técnicas como criar a base de dados no banco *SQLite* em memória ou mesmo usar o conceito de *Mocks* para emular a existência do banco de dados. Mas para nosso pequeno e didático projeto estes procedimentos servem para seu propósito.

Criando o teste para a entidade Post

Vamos iniciar criando o teste da entidade *Post*, que é salvo no arquivo *module/Application/tests/src/Application/Model/PostTest.php*:

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

use Core\Test\ModelTestCase;
use Application\Model\Post;
use Zend\InputFilter\InputFilterInterface;

/**
 * @group Model
 */
class PostTest extends ModelTestCase
{
    public function testGetInputFilter()
    {
          $post = new Post();
         $if = $post->getInputFilter();
            //testa se existem filtros
            $this->assertInstanceOf("Zend\InputFilter\InputFilter", $if);
            return $if;
        }
```

```
21
22          /**
23           * @depends testGetInputFilter
24           */
25          public function testInputFilterValid($if)
26          {
27              //testa os filtros
28              $this->assertEquals(4, $if->count());
29
30              $this->assertTrue($if->has('id'));
31              $this->assertTrue($if->has('title'));
32              $this->assertTrue($if->has('description'));
33              $this->assertTrue($if->has('post_date'));
34          }
35
36          /**
37           * @expectedException Core\Model\EntityException
38           */
39          public function testInputFilterInvalido()
40          {
41              //testa se os filtros estão funcionando
42              $post = new Post();
43              //title só pode ter 100 caracteres
44              $post->title = 'Lorem Ipsum e simplesmente uma simulacao de texto da i\
45  ndustria tipografica e de impressos. Lorem Ipsum é simplesmente uma simulac\
46  ao de texto da industria tipografica e de impressos';
47          }
48
49          /**
50           * Teste de insercao de um post valido
51           */
52          public function testInsert()
53          {
54              $post = $this->addPost();
55
56              $saved = $this->getTable('Application\Model\Post')->save($post);
57
58              //testa o filtro de tags e espacos
59              $this->assertEquals('A Apple compra a Coderockr', $saved->description)\
60  ;
61              //testa o auto increment da chave primaria
62              $this->assertEquals(1, $saved->id);
63          }
64
65          /**
66           * @expectedException Core\Model\EntityException
67           * @expectedExceptionMessage Input inválido: description =
68           */
69          public function testInsertInvalido()
```

```
70          {
71              $post = new Post();
72              $post->title = 'teste';
73              $post->description = '';
74
75              $saved = $this->getTable('Application\Model\Post')->save($post);
76          }
77
78          public function testUpdate()
79          {
80              $tableGateway = $this->getTable('Application\Model\Post');
81              $post = $this->addPost();
82
83              $saved = $tableGateway->save($post);
84              $id = $saved->id;
85
86              $this->assertEquals(1, $id);
87
88              $post = $tableGateway->get($id);
89              $this->assertEquals('Apple compra a Coderockr', $post->title);
90
91              $post->title = 'Coderockr compra a Apple';
92              $updated = $tableGateway->save($post);
93
94              $post = $tableGateway->get($id);
95              $this->assertEquals('Coderockr compra a Apple', $post->title);
96          }
97
98          /**
99           * @expectedException Core\Model\EntityException
100          * @expectedExceptionMessage Could not find row 1
101          */
102         public function testDelete()
103         {
104            $tableGateway = $this->getTable('Application\Model\Post');
105             $post = $this->addPost();
106
107             $saved = $tableGateway->save($post);
108             $id = $saved->id;
109
110             $deleted = $tableGateway->delete($id);
111             $this->assertEquals(1, $deleted); //numero de linhas excluidas
112
113             $post = $tableGateway->get($id);
114         }
115
116         private function addPost()
117         {
118             $post = new Post();
```

```
119             $post->title = 'Apple compra a Coderockr';
120             $post->description = 'A Apple compra a <b>Coderockr</b><br> ';
121             $post->post_date = date('Y-m-d H:i:s');
122 
123             return $post;
124         }
125 }
```

https://gist.github.com/4011992

Se esse foi seu primeiro contato com um teste automatizado escrito usando-se o framework *PHPUnit* imagino que você deve estar um pouco perdido e chocado com a quantia de código acima. Vou explicar os detalhes mais importantes.

```
1 namespace Application\Model;
2 use Core\Test\ModelTestCase;
3 use Application\Model\Post;
4 use Zend\InputFilter\InputFilterInterface;
```

O Zend Framework 2 usa extensivamente o conceito de *Namespaces* que foi introduzido no *PHP 5.3*. Na primeira linha indicamos a qual espaço o código que vamos escrever pertence, sendo obrigatório sempre termos um. Nas próximas linhas indicamos quais componentes vamos precisar e de quais *namespaces* eles pertencem. Fazendo isso as classes são automaticamente carregadas, sem precisarmos nos preocupar em usar *include* ou *require*, deixando essa tarefa para o mecanismo de *autoloading* implementado pelo framework. Mais informações sobre *namespaces* podem ser encontradas no manual do PHP.

```
1 /**
2 * @group Model
3 */
```

Indica a qual grupo de testes este teste pertence. Não é algo obrigatório, mas facilita a execução dos testes em grupos separados.

```
1 public function testGetInputFilter()
```

Responsável pelo teste da existência de um conjunto de filtros na entidade. Filtros e validadores serão usados para garantirmos a segurança da nossa aplicação. Veremos mais sobre eles mais tarde.

```
1 public function testInputFilterValid($if)
```

Enquanto o teste anterior verificava a existência de um conjunto de filtros, aqui testamos cada um dos campos que queremos filtrar. Por enquanto testamos se existe um filtro em específico para cada item da entidade.

```
1 public function testInputFilterInvalido()
```

Neste teste verificamos se o filtro do item *title* está funcionando. Nós queremos que o campo não tenha mais do que 100 caracteres.

```
1  public function testInsert()
```

Este teste é importante pois nos traz novos conceitos. O mais importante deles é o uso do *$this->getTable('Aplication\Model\Post')*. Esta chamada nos retorna uma instância de um *TableGateway* genérico que desenvolvi e está no módulo *Core*. A função de um *TableGateway* é realizar operações sobre entidades pois elas não possuem comportamento, sendo apenas representações dos dados. As principais operações como inserir, remover, atualizar, pesquisar serão sempre feitas através desse *gateway*. Existem muitas vantagens nessa abordagem como podermos mudar a camada das entidades (substituir por entidades do *ORM Doctrine* por exemplo) ou mudar o comportamento de todas as entidades apenas alterando o *gateway*. O *testInsert()* tenta salvar a entidade na tabela e verifica se obteve um *id*, que é gerado automaticamente pelo banco de dados (linha 61).

```
1  public function testInsertInvalido()
```

Simula uma inclusão inválida e verifica se é gerada uma *exception*, no caso uma *EntityException*. Se ela foi gerada significa que o validador do campo está funcionando corretamente.

```
1  public function testUpdate() e public function testDelete()
```

Estes dois testes testam a alteração e exclusão de um registro e não apresentam novos conceitos.

Rodando os testes pela primeira vez

Agora que criamos o nosso primeiro teste vamos rodar o comando *phpunit* para verificarmos o resultado:

```
1  phpunit -c module/Application/tests/phpunit.xml
2  PHPUnit 3.7.7 by Sebastian Bergmann.
3
4  Configuration read from /Users/eminetto/Documents/Projects/zf2napratica/mod\
5  ule/Application/tests/phpunit.xml
6
7  PHP Fatal error:  Class 'Application\Model\Post' not found in /Users/eminet\
8  to/Documents/Projects/zf2napratica/module/Application/tests/src/Application\
9  /Model/PostTest.php on line 15
```

Esse primeiro erro era esperado pois ainda não criamos a classe *Post*, o que faremos agora.

Criando a entidade Post

O primeiro passo é criar o diretório (caso não exista) de modelos:

```
1  mkdir module/Application/src/Application/Model
```

E criar dentro dele o arquivo *Post.php* com o conteúdo:

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

use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;
use Core\Model\Entity;

/**
* Entidade Post
*
* @category Application
* @package Model
*/
class Post extends Entity
{
    /**
    * Nome da tabela. Campo obrigatorio
    * @var string
    */
    protected $tableName ='posts';

    /**
    * @var int
    */
    protected $id;

    /**
    * @var string
    */
    protected $title;

    /**
    * @var string
    */
    protected $description;

    /**
    * @var datetime
    */
    protected $post_date;

    /**
    * Configura os filtros dos campos da entidade
    *
    * @return Zend\InputFilter\InputFilter
    */
    public function getInputFilter()
```

```
50      {
51          if (!$this->inputFilter) {
52              $inputFilter = new InputFilter();
53              $factory     = new InputFactory();
54
55              $inputFilter->add($factory->createInput(array(
56                  'name'     => 'id',
57                  'required' => true,
58                  'filters'  => array(
59                      array('name' => 'Int'),
60                  ),
61              )));
62
63              $inputFilter->add($factory->createInput(array(
64                  'name'     => 'title',
65                  'required' => true,
66                          'filters'  => array(
67                              array('name' => 'StripTags'),
68                              array('name' => 'StringTrim'),
69                          ),
70                          'validators' => array(
71                              array(
72                          'name'    => 'StringLength',
73                          'options' => array(
74                              'encoding' => 'UTF-8',
75                              'min'      => 1,
76                              'max'      => 100,
77                          ),
78                      ),
79                  ),
80              )));
81
82              $inputFilter->add($factory->createInput(array(
83                  'name'     => 'description',
84                  'required' => true,
85                  'filters'  => array(
86                      array('name' => 'StripTags'),
87                      array('name' => 'StringTrim'),
88                  ),
89              )));
90
91              $inputFilter->add($factory->createInput(array(
92                  'name'     => 'post_date',
93                  'required' => false,
94                  'filters'  => array(
95                      array('name' => 'StripTags'),
96                      array('name' => 'StringTrim'),
97                  ),
98              )));
```

```
99
100             $this->inputFilter = $inputFilter;
101         }
102         return $this->inputFilter;
103     }
104 }
```

https://gist.github.com/4011993

Como citei anteriormente, as entidades são filhas da *Core\Model\Entity* e são apenas representações dos dados da base de dados. Nesta classe descrevemos o nome da tabela (*$tableName*), os atributos (*$title* por exemplo) e os seus filtros (na *public function getInputFilter()*).

Alguns detalhes que podemos configurar quanto aos atributos da entidade:

- Se são obrigatórios ou não, usando o *required* com valor *true* ou *false* (exemplo: linha 58)
- Os filtros. Exemplos podem ser encontrados no filtro *Int* (linha 60), que transforma o campo em inteiro; *StripTags* (linha 68) que remove quaisquer *tags HTML/JavaScript* no texto; e *StringTrim* (linha 69) que remove espaços no começo e final do texto. Existem outros filtros que podem ser encontrados no manual do framework, além da possibilidade de criarmos outros.
- As validações. Um exemplo pode ser encontrado no *StringLength* (linhas 73 a 77) que valida se o campo possui a codificação de carecteres *UTF-8* e se o tamanho está entre 1 e 100. Caso algum desses parâmetros não for respeitado (por exemplo tentando salvar 101 caracteres) será gerada uma *EntityException* indicando que o campo está inválido. Existem outros validadores que podem ser encontrados no manual do framework, além da possibilidade de criarmos outros.

Ao centralizarmos a configuração dos filtros e validadores na entidade podemos reusá-los em outros pontos do projeto, como no uso de formulários ou até no acesso via uma *API*.

Rodando os testes novamente

Agora que temos o código da entidade *Post* podemos executar novamente os testes:

```
1 phpunit -c module/Application/tests/phpunit.xml
2 PHPUnit 3.7.7 by Sebastian Bergmann.
3 Configuration read from /Users/eminetto/Documents/Projects/zf2napratica/mod\
4 ule/Application/tests/phpunit.xml
5 .......
6 Time: 3 seconds, Memory: 18.50Mb
7 OK (7 tests, 17 assertions)
8 Generating code coverage report in Clover XML format ... done
9 Generating code coverage report in HTML format ... done
```

E agora os testes devem funcionar. Caso algum teste falhe será indicado qual foi e o provável motivo da falha, auxiliando na correção.

Criando o teste da entidade Comment

Vamos agora criar o teste da entidade *Comment*, no arquivo *module/Application/tests/src/Model/CommentTest.php*:

```
<?php
namespace Application\Model;

use Core\Test\ModelTestCase;
use Application\Model\Post;
use Application\Model\Comment;
use Zend\InputFilter\InputFilterInterface;

/**
 * @group Model
 */
class CommentTest extends ModelTestCase
{
        public function testGetInputFilter()
        {
                $comment = new Comment();
                $if = $comment->getInputFilter();

                $this->assertInstanceOf("Zend\InputFilter\InputFilter", $if);
                return $if;
        }

        /**
        * @depends testGetInputFilter
        */
        public function testInputFilterValid($if)
        {
                $this->assertEquals(7, $if->count());

                $this->assertTrue(
                    $if->has('id')
                );
                $this->assertTrue(
                    $if->has('post_id')
                );
                $this->assertTrue(
                    $if->has('description')
                );
                $this->assertTrue(
                    $if->has('name')
                );
                $this->assertTrue(
                    $if->has('email')
                );
```

```
45                  $this->assertTrue(
46                      $if->has('webpage')
47                  );
48                  $this->assertTrue(
49                      $if->has('comment_date')
50                  );
51          }
52
53          /**
54          * @expectedException Core\Model\EntityException
55          * @expectedExceptionMessage Input inválido: email =
56          */
57          public function testInputFilterInvalido()
58          {
59                  $comment = new Comment();
60                  //email deve ser um e-mail válido
61                  $comment->email = 'email_invalido';
62          }
63
64          /**
65          * Teste de insercao de um comment valido
66          */
67          public function testInsert()
68          {
69                  $comment = $this->addComment();
70                  $saved = $this->getTable('Application\Model\Comment')->save($comment);
71                  $this->assertEquals(
72                          'Comentário importante alert("ok");', $saved->description
73                  );
74                  $this->assertEquals(1, $saved->id);
75          }
76
77          /**
78          * @expectedException Zend\Db\Adapter\Exception\InvalidQueryException
79          */
80          public function testInsertInvalido()
81          {
82                  $comment = new Comment();
83                  $comment->description = 'teste';
84                  $comment->post_id = 0;
85                  $saved = $this->getTable('Application\Model\Comment')->save($comment);
86                  }
87
88          public function testUpdate()
89          {
90                  $tableGateway = $this->getTable('Application\Model\Comment');
91                  $comment = $this->addComment();
92                  $saved = $tableGateway->save($comment);
93                  $id = $saved->id;
```

```
94
95                  $this->assertEquals(1, $id);
96
97                  $comment = $tableGateway->get($id);
98                  $this->assertEquals(
99                      'eminetto@coderockr.com', $comment->email
100                 );
101
102                 $comment->email = 'eminetto@gmail.com';
103                 $updated = $tableGateway->save($comment);
104
105                 $comment = $tableGateway->get($id);
106                 $this->assertEquals('eminetto@gmail.com', $comment->email);
107         }
108
109         /**
110         * @expectedException Zend\Db\Adapter\Exception\InvalidQueryException
111         * @expectedExceptionMessage Statement could not be executed
112         */
113         public function testUpdateInvalido()
114         {
115                 $tableGateway = $this->getTable('Application\Model\Comment');
116                 $comment = $this->addComment();
117                 $saved = $tableGateway->save($comment);
118                 $id = $saved->id;
119                 $comment = $tableGateway->get($id);
120                 $comment->post_id = 10;
121                 $updated = $tableGateway->save($comment);
122                 }
123
124         /**
125         * @expectedException Core\Model\EntityException
126         * @expectedExceptionMessage Could not find row 1
127         */
128         public function testDelete()
129         {
130                 $tableGateway = $this->getTable('Application\Model\Comment');
131                 $comment = $this->addComment();
132                 $saved = $tableGateway->save($comment);
133                 $id = $saved->id;
134
135                 $deleted = $tableGateway->delete($id);
136                 $this->assertEquals(1, $deleted); //numero de linhas excluidas
137
138                 $comment = $tableGateway->get($id);
139         }
140
141         private function addPost()
142         {
```

```
143                 $post = new Post();
144                 $post->title = 'Apple compra a Coderockr';
145                 $post->description = 'A Apple compra a <b>Coderockr</b><br> ';
146                 $post->post_date = date('Y-m-d H:i:s');
147 
148                 $saved = $this->getTable('Application\Model\Post')->save($post);
149                 return $saved;
150                 }
151 
152         private function addComment()
153         {
154                 $post = $this->addPost();
155                 $comment = new Comment();
156                 $comment->post_id = $post->id;
157                 $comment->description = 'Comentário importante <script>alert("ok");</scri\
158 pt> <br> ';
159                 $comment->name = 'Elton Minetto';
160                 $comment->email = 'eminetto@coderockr.com';
161                 $comment->webpage = 'http://www.eltonminetto.net';
162                 $comment->comment_date = date('Y-m-d H:i:s');
163                 return $comment;
164         }
165 }
```

https://gist.github.com/4011994

Criando o código da entidade Comment

Se rodarmos os testes agora vamos enfrentar aquele erro devido a não existência da classe *Comment*. Então vamos agora criar o código da entidade e salvar no arquivo *module/Application/src/Application/-Model/Comment.php*:

```
1  <?php
2  namespace Application\Model;
3 
4  use Zend\InputFilter\Factory as InputFactory;
5  use Zend\InputFilter\InputFilter;
6  use Zend\InputFilter\InputFilterAwareInterface;
7  use Zend\InputFilter\InputFilterInterface;
8  use Core\Model\Entity;
9 
10 /**
11 * Entidade Comment
12 *
13 * @category Application
14 * @package Model
15 */
16 class Comment extends Entity
```

```
17  {
18
19      /**
20      * Nome da tabela. Campo obrigatório
21      * @var string
22      */
23      protected $tableName ='comments';
24
25      /**
26      * @var int
27      */
28      protected $id;
29
30      /**
31      * @var int
32      */
33      protected $post_id;
34
35      /**
36      * @var string
37      */
38      protected $description;
39
40      /**
41      * @var string
42      */
43      protected $name;
44
45      /**
46      * @var string
47      */
48      protected $email;
49
50      /**
51      * @var string
52      */
53      protected $webpage;
54
55      /**
56      * @var datetime
57      */
58      protected $comment_date;
59
60      /**
61      * Configura os filtros dos campos da entidade
62      *
63      * @return Zend\InputFilter\InputFilter
64      */
65      public function getInputFilter()
```

```
66      {
67          if (!$this->inputFilter) {
68              $inputFilter = new InputFilter();
69              $factory     = new InputFactory();
70
71              $inputFilter->add($factory->createInput(array(
72                  'name'     => 'id',
73                  'required' => true,
74                      'filters'  => array(
75                          array('name' => 'Int'),
76                      ),
77              )));
78
79              $inputFilter->add($factory->createInput(array(
80                  'name'     => 'post_id',
81                  'required' => true,
82                  'filters'  => array(
83                      array('name' => 'Int'),
84                  ),
85              )));
86
87              $inputFilter->add($factory->createInput(array(
88                  'name'     => 'description',
89                  'required' => true,
90                  'filters'  => array(
91                      array('name' => 'StripTags'),
92                      array('name' => 'StringTrim'),
93                  ),
94              )));
95
96              $inputFilter->add($factory->createInput(array(
97                  'name'     => 'email',
98                  'required' => true,
99                  'filters'  => array(
100                     array('name' => 'StripTags'),
101                     array('name' => 'StringTrim'),
102                 ),
103                 'validators' => array(
104                     array(
105                         'name'    => 'EmailAddress',
106                     ),
107                 ),
108             )));
109
110             $inputFilter->add($factory->createInput(array(
111                 'name'     => 'name',
112                 'required' => true,
113                 'filters'  => array(
114                     array('name' => 'StripTags'),
```

```
115                     array('name' => 'StringTrim'),
116                 ),
117                 'validators' => array(
118                     array(
119                         'name'    => 'StringLength',
120                         'options' => array(
121                             'encoding' => 'UTF-8',
122                             'min'      => 1,
123                             'max'      => 100,
124                         ),
125                     ),
126                 ),
127             )));
128 
129             $inputFilter->add($factory->createInput(array(
130                 'name'     => 'webpage',
131                 'required' => true,
132                 'filters'  => array(
133                     array('name' => 'StripTags'),
134                     array('name' => 'StringTrim'),
135                 ),
136                 'validators' => array(
137                     array(
138                         'name'    => 'StringLength',
139                         'options' => array(
140                             'encoding' => 'UTF-8',
141                             'min'      => 1,
142                             'max'      => 200,
143                         ),
144                     ),
145                 ),
146             )));
147 
148             $inputFilter->add($factory->createInput(array(
149                 'name'     => 'comment_date',
150                 'required' => false,
151                 'filters'  => array(
152                     array('name' => 'StripTags'),
153                     array('name' => 'StringTrim'),
154                 ),
155             )));
156 
157             $this->inputFilter = $inputFilter;
158         }
159         return $this->inputFilter;
160     }
161 }
```

https://gist.github.com/4011996

A principal novidade nestes códigos é o uso do validador *EmailAddress* que verifica se o valor informado é um endereço válido. O restante dos códigos seguem o que foi explicado na entidade *Post*.

Podemos agora executar novamente os testes para verificar se todos estão passando.

A entidade *User* será criada nos próximos capítulos, quando criarmos o módulo de administração do sistema.

Queries

Neste capítulo vimos como manipular uma entidade usando o conceito de um *TableGateway* mas em alguns casos precisamos criar consultas mais complexas, como fazer *join* entre tabelas ou criar *subqueries*. Para facilitar a criação de consultas que funcionem em diversos bancos de dados o framework fornece componentes, dentro do *namespace Zend\Db\Sql*.

Vamos ver alguns exemplos.

O primeiro passo que precisamos fazer é indicar o uso do componente:

```
1  use Zend\Db\Sql\Sql;
```

A construção do objeto *Sql* depende de uma conexão com o banco de dados. Para isso vamos usar a conexão configurada no projeto:

```
1  $adapter = $this->getServiceLocator()->get('DbAdapter');
```

Obs: veremos com detalhes o que significa o *getServiceLocator()* no capítulo sobre serviços. Por enquanto nos basta saber que essa linha de código retorna a conexão atual com o banco de dados.

Consulta simples, como um *SELECT * FROM posts*

```
1  $sql = new Sql($adapter);
2  $select = $sql->select()->from('posts');
3  $statement = $sql->prepareStatementForSqlObject($select);
4  $results = $statement->execute();
5  foreach ($results as $r) {
6          echo $r['id'], ' - ', $r['title'], '<br>';
7  }
```

Consulta com clausula WHERE, como um *SELECT * FROM posts WHERE id = 10*:

```php
$sql = new Sql($adapter);
$select = $sql->select()
        ->from('posts')
        ->where(array('id' => 10));

$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
        echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Consulta com seleção de campos e cláusula WHERE, como um *SELECT id, title FROM posts WHERE id > 10*:

Neste caso podemos usar de duas formas:

```php
$sql = new Sql($adapter);
$select = $sql->select()
        ->columns(array('id','title'))
        ->from('posts')
        ->where('id > 10');
$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
        echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Ou usando o componente *Predicate*. Primeiro precisamos importar o componente:

```php
use Zend\Db\Sql\Predicate\Predicate;
```

E usamos da seguinte forma:

```php
$sql = new Sql($adapter);
$select = $sql->select()
        ->columns(array('id','title'))
        ->from('posts');

$where = new Predicate();
$where->greaterThan('id',10);
$select->getRawState($select::WHERE)->addPredicate($where);
$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
        echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Consulta com seleção de campos e cláusula WHERE e INNER JOIN, como um *SELECT posts. id, posts. title, comments. description, comments.name FROM posts INNER JOIN comments ON comments.post_id = post.id WHERE posts.id > 10* order by title:

```php
$sql = new Sql($adapter);
$select = $sql->select();
$select->columns(array('id','title'))
        ->from('posts')
        ->join(
                'comments',
                'comments.post_id = posts.id',
                array('description'),
                $select::JOIN_INNER
        )
        ->order(array('title'));

$where = new Predicate();
$where->greaterThan('posts.id',10);

$select->getRawState($select::WHERE)->addPredicate($where);

$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
        echo $r['id'], ' - ', $r['title'], '-', $r['name'], ' - ', $r['description\
'] . '<br>';
}
```

Em qualquer momento é possível imprimir a consulta que foi gerada, o que pode ser útil para fins de debug:

```php
echo $select->getSqlString();
```

Mais detalhes no manual do framework

Vamos agora trabalhar com a próxima camada, os controladores.

Controladores

Os controladores são responsáveis pela interação com o usuário, interceptando as invocações por urls, cliques em links, etc. O controlador irá receber a ação do usuário, executar algum serviço, manipular alguma entidade e, geralmente, renderizar uma tela da camada de visão, para gerar uma resposta para o usuário.

O primeiro controlador que iremos criar é o responsável por mostrar todos os posts de nossa tabela, para que o usuário os visualize.

Para criar um controlador precisamos escrever uma classe que implemente a interface *Dispatchable*. O Zend Framework fornece algumas classes que implementam esta interface, facilitando o nosso uso, como a *AbstractActionController* e a *AbstractRestfulController*. No nosso exemplo vamos estender uma classe que consta no módulo *Core* a *Core\Controller\ActionController* que irá nos fornecer algumas funcionalidades adicionais, como o método *getTable()* que foi comentado no capítulo anterior.

Criando os testes

Novamente iniciaremos o processo criando o teste para o nosso controlador. Assim podemos pensar em suas funcionalidades antes de iniciarmos a codificação. Primeiro precisamos criar o diretório de testes dos controladores:

```
1  mkdir  module/Application/tests/src/Application/Controller
```

Vamos agora criar o arquivo de testes, no arquivo *module/Application/tests/src/Application/Controller/IndexControllerTest.php*

```
1  <?php
2
3  use Core\Test\ControllerTestCase;
4  use Application\Controller\IndexController;
5  use Application\Model\Post;
6  use Zend\Http\Request;
7  use Zend\Stdlib\Parameters;
8  use Zend\View\Renderer\PhpRenderer;
9
10
11 /**
12  * @group Controller
13 */
14 class IndexControllerTest extends ControllerTestCase
15 {
16     /**
17     * Namespace completa do Controller
18     * @var string
19     */
20     protected $controllerFQDN = 'Application\Controller\IndexController';
21
```

```
22      /**
23      * Nome da rota. Geralmente o nome do modulo
24      * @var string
25      */
26      protected $controllerRoute = 'application';
27
28      /**
29      * Testa o acesso a uma action que nao existe
30      */
31      public function test404()
32      {
33          $this->routeMatch->setParam('action', 'action_nao_existente');
34          $result = $this->controller->dispatch($this->request);
35          $response = $this->controller->getResponse();
36          $this->assertEquals(404, $response->getStatusCode());
37      }
38
39      /**
40      * Testa a pagina inicial, que deve mostrar os posts
41      */
42      public function testIndexAction()
43      {
44          // Cria posts para testar
45          $postA = $this->addPost();
46          $postB = $this->addPost();
47
48          // Invoca a rota index
49          $this->routeMatch->setParam('action', 'index');
50          $result = $this->controller->dispatch(
51              $this->request, $this->response
52          );
53
54          // Verifica o response
55          $response = $this->controller->getResponse();
56          $this->assertEquals(200, $response->getStatusCode());
57
58          // Testa se um ViewModel foi retornado
59          $this->assertInstanceOf(
60              'Zend\View\Model\ViewModel', $result
61          );
62
63          // Testa os dados da view
64          $variables = $result->getVariables();
65          $this->assertArrayHasKey('posts', $variables);
66
67          // Faz a comparação dos dados
68          $controllerData = $variables["posts"];
69          $this->assertEquals(
70              $postA->title, $controllerData[0]['title']
```

```
71          );
72          $this->assertEquals(
73              $postB->title, $controllerData[1]['title']
74          );
75      }
76
77      /**
78      * Adiciona um post para os testes
79      */
80      private function addPost()
81      {
82          $post = new Post();
83          $post->title = 'Apple compra a Coderockr';
84          $post->description = 'A Apple compra a <b>Coderockr</b><br> ';
85          $post->post_date = date('Y-m-d H:i:s');
86          $saved = $this->getTable('Application\Model\Post')->save($post);
87          return $saved;
88      }
89  }
```

https://gist.github.com/4012000

Vamos aos detalhes do código.

```
1  /**
2  * @group Controller
3  */
```

Similar aos testes de entidades, indicamos que este teste pertence a um grupo: o *Controller*. Assim podemos executar somente os testes deste grupo caso necessário.

```
1  class IndexControllerTest extends ControllerTestCase
```

Todos os testes de controladores devem estender a classe *ControllerTestCase*.

```
1  protected $controllerFQDN = 'Application\Controller\IndexController';
```

Namespace do controlador. É necessária para os testes.

```
1  protected $controllerRoute = 'application';
```

Nome da rota a ser usada. Geralmente é o mesmo nome do módulo, com letras minúsculas. Voltaremos a falar sobre as rotas no capítulo de criação de novos módulos.

```
1 public function test404()
```

Este teste verifica o que acontece caso o usuário tente acessar uma *action* não existente.

Vamos abrir um pequeno parênteses aqui, e comentar sobre controladores e actions. Geralmente os controladores são classes com o nome terminando em *Controller* como o *IndexController* (apesar disso não ser mais obrigatório no Zend Framework 2 ainda continua se usando esse padrão). Cada controlador possui uma ou mais *actions* que são métodos públicos cujo nome termina com *Action* como o *indexAction*. As *actions* são as ações que os usuários podem acessar via URL, links ou botões na tela. Por exemplo, caso o usuário acesse a url:

```
1 http://zf2napratica.dev/application/index/index/id/1
```

Isto é traduzido pelo framework usando o padrão:

```
1 http://servidor/modulo/controller/action/parametro/valor
```

Então:

- Servidor = *zf2napratica.dev*
- Módulo = *Application*
- Controller = *IndexController.php*
- Action = *indexAction* (dentro do arquivo *IndexController.php*)
- Parâmetro = *id*
- Valor = 1

Caso o usuário tente acessar uma *action* que não existe um erro é gerado, exatamente o que o teste *test404()* verifica.

```
1 public function testIndexAction()
```

Este é o teste mais importante. Primeiro criamos dois registros na tabela *Post* (*addPost*), depois simulamos o acesso a *action index* (*$this->routeMatch->setParam('action', 'index');*). O comando *dispatch* executa o acesso efetivamente, como se fosse um usuário acessando a URL no navegador. Após verificar se a página foi encontrada (*statusCode* 200) verificamos se recebemos uma instância de *ViewModel* e se ela possui os dados dos *posts*. A *ViewModel* é a representação em forma de objeto da nossa camada de visão, que será renderizada na forma de *HTML* no navegador. Iremos criar a visão ainda neste capítulo.

Agora podemos executar nossos testes novamente. Podemos executar somente os testes do grupo *Controller* para que o teste seja executado mais rapidamente:

```
1 phpunit -c module/Application/tests/phpunit.xml  --group=Controller
```

Os testes irão falhar pois ainda não criamos o controlador e sua visão.

Criando o controlador

Vamos criar o controlador, no arquivo *module/Application/src/Application/Controller/IndexController.php*:

```php
<?php
namespace Application\Controller;

use Zend\View\Model\ViewModel;
use Core\Controller\ActionController;

/**
 * Controlador que gerencia os posts
 *
 * @category Application
 * @package Controller
 * @author  Elton Minetto <eminetto@coderockr.com>
 */
class IndexController extends ActionController
{
    /**
    * Mostra os posts cadastrados
    * @return void
    */
    public function indexAction()
    {
        return new ViewModel(array(
            'posts' => $this->getTable('Application\Model\Post')
                            ->fetchAll()
                            ->toArray()
        ));
    }
}
```

https://gist.github.com/4012002

Como citado anteriormente, a class *IndexController* estende a *ActionController*, o que nos dá acesso ao método *getTable()*. Com resultado do *getTable()* (que é uma instância de *TableGateway*) podemos usar o método *fetchAll()* para recuperar todos os registros da tabela, e o *toArray()* para convertê-los em *arrays*. A *indexAction* cria uma instância da classe *ViewModel* e a configura com uma variável chamada *posts* com o resultado da consulta com a base de dados. Ao retornar a instância de *ViewModel* estamos solicitando ao framework que renderize o *HTML* da visão. Como não indicamos alguma visão em específico a visão padrão será renderizada. Como estamos acessando a *action index* do controlador *IndexController* o framework irá procurar um arquivo chamado *index.phtml* no diretório: *module/Application/view/application/index/*.

Visões

Apesar da extensão diferente (*.phtml*) a visão não passa de um arquivo *PHP* onde podemos usar todas as funções nativas da linguagem e mais alguns componentes especiais, chamados de *View Helpers*. Veremos

mais detalhes sobre os *ViewHelpers* em um próximo capítulo.

O conteúdo do arquivo *index.phtml* é:

```
<div class="actions clearfix">
    <div class="btns">
        <a class="btn submit" href="/admin/index/save" title="Criar Post">
            Criar Post
        </a>
    </div>
</div>
<label class="divisor"><span>Lista de Posts</span></label>
<table class="datatable">
    <thead>
        <tr>
            <th>Título</th>
            <th>Texto</th>
            <th width="130" class="center">Data de Cadastro</th>
            <th width="120" class="center">Opções</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach($posts as $post):?>
        <tr>
            <td><?php echo $this->escapeHtml($post['title']);?></td>
            <td><?php echo $this->escapeHtml($post['description']);?></td>
            <td class="center">
            <?php
            echo $this->dateFormat(
                $post['post_date'],
                \IntlDateFormatter::SHORT,
                \IntlDateFormatter::SHORT,
                'pt_BR'
            );
            ?>
            </td>
            <td class="center">
                <a href="/admin/index/save/id/<?php echo $post['id'] ;?>"
                    title="Editar" class="btn">
                    <i class="icon-edit"></i>
                </a>
                <a href="/admin/index/delete/id/<?php echo $post['id'];?>"
                    rel="confirmation"
                    title="Deseja excluir este registro?"
                    class="btn">
                    <i class="icon-remove"></i>
                </a>
            </td>
        </tr>
        <?php endforeach;?>
```

```
47      </tbody>
48  </table>
```

https://gist.github.com/4012005

Alguns pontos importantes do *index.phtml*:

```
1  <?php foreach($posts as $post):?>
```

A variável *posts* é a mesma que foi enviada pelo controlador, ou seja, a lista de registros vindos da tabela *post*.

```
1  <?php echo $this->escapeHtml($post['title']);?>
```

Este é um exemplo de uso de um *ViewHelper*, o *escapeHtml*, que limpa quaisquer *tags HTML* que possam existir no texto. O mesmo vale para o *dateFormat* que faz a formatação das datas.

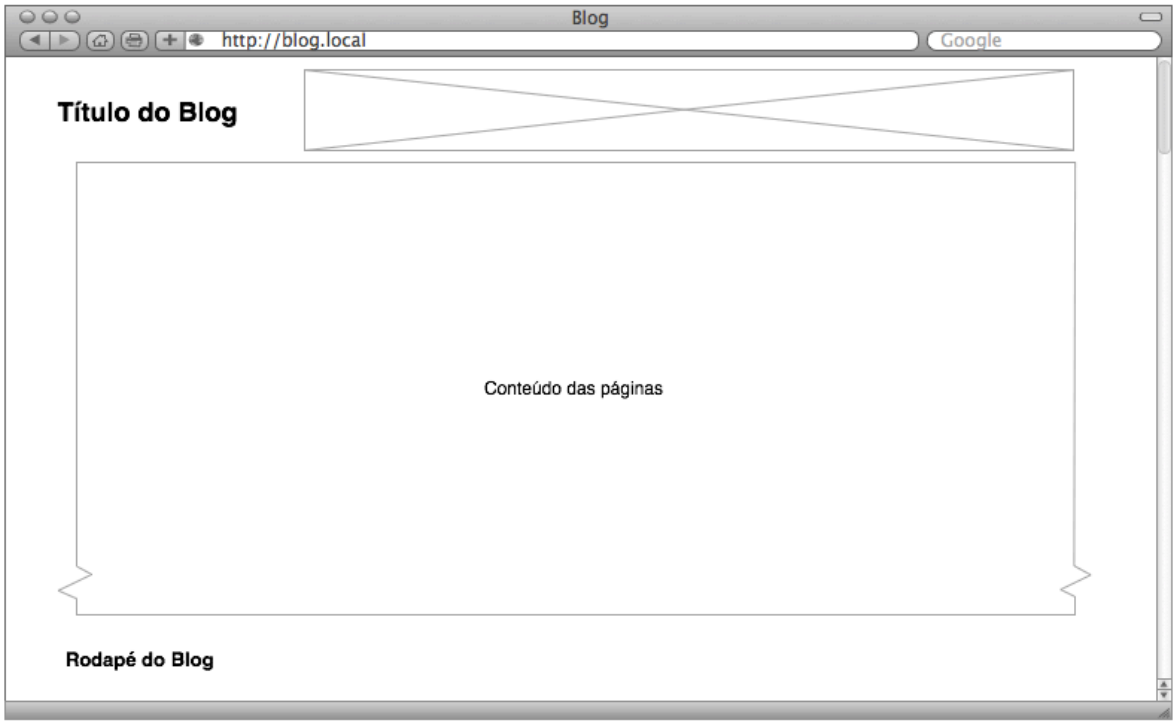
```
1  <a class="btn submit" href="/admin/index/save" title="Criar Post">
2          Criar Post
3  </a>
```

Estamos criando links para acessar um outro controlador, que criaremos no próximo capítulo, o *IndexController* do módulo *Admin*.

Layouts

Neste momento cabe explicar um novo conceito: *layouts*. O Zend Framework trabalha com a idéia de *layouts*, que são molduras ou páginas padrão mescladas com as visões dos controladores, como a *index.phtml*.

Usando uma imagem para ilustrar um exemplo:



Wireframe mostrando o layout

Em todas as páginas teremos um título, uma imagem e um rodapé, com informações sobre o autor, *copyright*, etc. A única informação que mudará é o conteúdo das páginas, como o *index.phtml* que acabamos de criar, mas o cabeçalho e o rodapé permanecem os mesmos.

Podemos ter vários *layouts* e usarmos o mais indicado em cada controlador ou *action*. O código do *layout* padrão do Zend Framework 2 está no arquivo *module/Application/view/layout/layout.phtml*:

```
1  <?php echo $this->doctype(); ?>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <?php
6          echo $this->headTitle('ZF2 '. $this->translate('Skeleton Applicatio\
7  n'))
8                      ->setSeparator(' - ')
9                      ->setAutoEscape(false)
10     ?>
11     <?php echo $this->headMeta()->appendName('viewport', 'width=device-widt\
12 h, initial-scale=1.0') ?>
13     <!-- Styles -->
14     <?php
15         echo $this->headLink(
16                         array(
17                             'rel' => 'shortcut icon',
18                             'type' => 'image/vnd.microsoft.icon',
19                             'href' => $this->basePath() . '/images/favicon.\
```

```
ico'
                        )
                    )
                ->prependStylesheet($this->basePath() . '/css/bootstrap-r\
esponsive.min.css')
                ->prependStylesheet($this->basePath() . '/css/style.css')
                ->prependStylesheet($this->basePath() . '/css/bootstrap.m\
in.css')
    ?>

    <!-- Scripts -->
    <?php
      echo $this->headScript()
                ->prependFile(
                    $this->basePath() . '/js/html5.js',
                    'text/javascript',
                    array('conditional' => 'lt IE 9',)
                )
                ->prependFile($this->basePath() . '/js/jquery-1.7.2.min.js'\
)
    ?>
    </head>
<body>
    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
            <a class="btn btn-navbar" data-toggle="collapse" data-target=".\
nav-collapse">
                 <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </a>
            <a class="brand" href="/">
              <?php echo $this->translate('Skeleton Application') ?>
            </a>
            <div class="nav-collapse">
                <ul class="nav">
                    <li class="active">
                        <a href="/">
                          <?php echo $this->translate('Home') ?>
                        </a>
                    </li>
                    <li><a href="/admin/auth/index">Entrar</a></li>
                </ul>
            </div><!--/.nav-collapse -->
        </div>
    </div>
</div>
<div class="container">
```

```
69      <?php echo $this->content; ?>
70      <hr>
71      <footer>
72          <p>
73            &copy; 2005 - 2012 by Zend Technologies Ltd.
74            <?php echo $this->translate('All rights reserved.') ?>
75          </p>
76      </footer>
77      </div> <!-- /container -->
78      <?php echo $this->inlineScript() ?>
79      </body>
80  </html>
```

https://gist.github.com/4012012

Este arquivo faz uso de diversos *ViewHelpers* que não iremos ver agora, como o *headLink* ou o *translate*, e possui uma estrutura bem complexa de design e *CSS*. Mas a parte mais importante para nós agora é a linha:

```
1  <?php echo $this->content; ?>
```

É nesse ponto que a visão será renderizada no *layout*, ou seja, o conteúdo do *index.phtml* vai ser mesclado com o restante. Como todos os controladores usam esse *layout*, por padrão esse comportamento será igual em todos, a menos que o desenvolvedor altere a configuração de algum deles. O layout pode ser muito simples, apenas precisa constar essa linha, que imprime o *$this->content*.

Executando os testes novamente

Executando os testes novamente todos devem passar. Caso algum falhe é preciso revisar os códigos e configurações que vimos nesse capítulo.

No próximo capítulo vamos incrementar nosso controlador com um paginador.

Desafio

Neste capítulo vimos o teste e o controlador para manipular a entidade *Post*. Como desafio deixo ao leitor a inclusão de uma nova funcionalidade: a listagem de comentários do post. Lembre-se de alterar o *IndexControllerTest* para incluir o novo teste que irá verificar a existência dos comentários, alterar o *IndexController* e também a *view* para mostrá-los.

Os códigos desse desafio estão disponíveis no repositório do *Github*:

https://github.com/eminetto/zf2napratica

Você também pode contribuir com exemplos fazendo um *fork* do repositório e colocando sua solução.

Paginador

A *indexAction* que criamos no *IndexController* possui um problema sério. Atualmente ela faz a busca de todos os registros da tabela *post* e simplesmente as envia para a *ViewModel* renderizar. Mas o que acontece se tivermos milhares de registros na tabela? Um sério problema de performance, pois o usuário irá esperar por vários minutos até a página ser renderizada (isso se o servidor não cortar a transmissão antes). Para resolver esse problema usaremos o componente *Zend\Paginator*.

O *Paginator* facilita a paginação da informação e foi desenvolvido com os seguintes princípios:

- Paginar qualquer tipo de dados, não apenas o resultado de banco de dados. Podemos paginar *arrays* ou classes que implementem a interface *Iterator* do *SPL*.
- Recuperar apenas os resultados necessários para a visualização, aumentando a performance.
- Ser independente dos outros componentes do framework para facilitar ao desenvolvedor usá-lo inclusive em outros projetos.

Adicionando o teste do paginador

Vamos adicionar um novo teste ao nosso arquivo *module/Application/tests/src/Application/Controller/IndexControllerTest.php* e fazer uma pequena alteração no *testIndexAction()*:

```
1  /**
2  * Testa a pagina inicial, que deve mostrar os posts
3  */
4  public function testIndexAction()
5  {
6          // Cria posts para testar
7          $postA = $this->addPost();
8          $postB = $this->addPost();
9
10         // Invoca a rota index
11         $this->routeMatch->setParam('action', 'index');
12         $result = $this->controller->dispatch($this->request, $this->response);
13
14         // Verifica o response
15         $response = $this->controller->getResponse();
16         $this->assertEquals(200, $response->getStatusCode());
17
18         // Testa se um ViewModel foi retornado
19         $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
20
21         // Testa os dados da view
22         $variables = $result->getVariables();
23         $this->assertArrayHasKey('posts', $variables);
24
25         // Faz a comparação dos dados
26         //mudamos a linha abaixo
27         $controllerData = $variables["posts"]->getCurrentItems()->toArray();
```

```
28          $this->assertEquals($postA->title, $controllerData[0]['title']);
29          $this->assertEquals($postB->title, $controllerData[1]['title']);
30  }
31
32  /**
33  * Testa a pagina inicial, que deve mostrar os posts com paginador
34  */
35  public function testIndexActionPaginator()
36  {
37          // Cria posts para testar
38          $post = array();
39          for($i=0; $i< 25; $i++) {
40                  $post[] = $this->addPost();
41          }
42
43          // Invoca a rota index
44          $this->routeMatch->setParam('action', 'index');
45          $result = $this->controller->dispatch($this->request, $this->response);
46
47          // Verifica o response
48          $response = $this->controller->getResponse();
49          $this->assertEquals(200, $response->getStatusCode());
50
51          // Testa se um ViewModel foi retornado
52          $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
53
54          // Testa os dados da view
55          $variables = $result->getVariables();
56
57          $this->assertArrayHasKey('posts', $variables);
58
59          //testa o paginator
60          $paginator = $variables["posts"];
61          $this->assertEquals(
62              'Zend\Paginator\Paginator', get_class($paginator)
63          );
64          $posts = $paginator->getCurrentItems()->toArray();
65          $this->assertEquals(10, count($posts));
66          $this->assertEquals($post[0]->id, $posts[0]['id']);
67          $this->assertEquals($post[1]->id, $posts[1]['id']);
68
69          //testa a terceira pagina da paginacao
70          $this->routeMatch->setParam('action', 'index');
71          $this->routeMatch->setParam('page', 3);
72          $result = $this->controller
73                          ->dispatch($this->request, $this->response);
74          $variables = $result->getVariables();
75          $controllerData = $variables["posts"]->getCurrentItems()->toArray();
76          $this->assertEquals(5, count($controllerData));
```

```
77 }
```

https://gist.github.com/4012024

Além de testarmos se a *ViewModel* agora possui um objeto do tipo *Zend\Paginator\Paginator* verificamos se foi mostrado apenas 10 registros por página (o valor padrão do paginador) e o que acontece quando acessamos a página 3 (*$this->routeMatch->setParam('page', 3);*), o que seria equivalente a acessar a URL:

```
1 http://zf2napratica.dev/application/index/index/page/3
```

Se executarmos os testes agora eles devem falhar, pois ainda não adicionamos o paginador no nosso controlador.

Adicionando o paginador no IndexController

Precisamos incluir novos *namespaces* no início do arquivo, para usarmos o paginador:

```
1 use Zend\Paginator\Paginator;
2 use Zend\Paginator\Adapter\DbSelect as PaginatorDbSelectAdapter;
```

Agora vamos alterar a *indexAction*:

```
1  /**
2   * Mostra os posts cadastrados
3  * @return void
4  */
5  public function indexAction()
6  {
7          $post = $this->getTable('Application\Model\Post');
8          $sql = $post->getSql();
9          $select = $sql->select();
10
11         $paginatorAdapter = new PaginatorDbSelectAdapter($select, $sql);
12         $paginator = new Paginator($paginatorAdapter);
13         $paginator->setCurrentPageNumber($this->params()->fromRoute('page'));
14
15         return new ViewModel(array(
16                 'posts' => $paginator
17         ));
18 }
```

https://gist.github.com/4012025

Vamos aos detalhes do código.

```
$sql = $post->getSql();
$select = $sql->select();
```

Como vamos paginar informações vindas do banco de dados precisamos de um objeto *select* e um *sql*, que são necessários para o paginador funcionar. Felizmente o *TableGateway* fornece esses componentes, o que facilita o acesso.

```
$paginatorAdapter = new PaginatorDbSelectAdapter($select, $sql);
$paginator = new Paginator($paginatorAdapter);
```

Conforme comentado no início do capítulo, o paginador é genérico, então precisamos passar como parâmetro o que iremos paginar, nesse caso um *DbSelect* (que eu apelidei de *PaginatorDbSelectAdapter* na importação do *namespace*).

```
$paginator->setCurrentPageNumber($this->params()->fromRoute('page'));
```

Essa configuração diz ao paginador em qual página ele se encontra. Caso não seja indicada nenhuma página pela URL (*page/3* por exemplo) a primeira página é apresentada.

Precisamos agora alterar a visão para que ela mostre o paginador.

Adicionamos o código abaixo no arquivo *module/Application/view/application/index/index.phtml*:

```
<?php
echo $this->paginationControl(
        $posts,
        'Sliding',
        'partials/paginator/control.phtml'
);
?>
```

Estamos imprimindo o controlador de paginadores e passando como parâmetros o paginador (*$posts*), o formato (*Sliding*, similar ao paginador da página de busca do *Google*) e qual é o trecho de código que contém os links (*próxima página*, *página anterior*, etc).

O conteúdo do terceiro parâmetro é algo novo para nós.

Partials

No terceiro parâmetro da chamada ao *paginationControl* indicamos o *partial* onde consta o código dos links do paginador. *Partials* são porções de código que podemos salvar e usar em diversas visões e *layouts*. Isso ajuda bastante a organização de código, pois podemos compartilhar um trecho de *HTML* ou *PHP* entre diversas visões.

Precisamos criar o diretório de *partials* do módulo:

```
mkdir module/Application/view/partials
```

E criar o diretório para armazenar os *partials* do paginador:

```
mkdir module/Application/view/partials/paginator
```

O código do arquivo *mkdir module/Application/view/partials/paginator/control.phtml* é:

```
<?php if ($this->pageCount > 1): ?>
<div class="navigation clearfix">
<?php if($this->current != 1): ?>
        <a href="<?= $this->url(null, array('page' => 1), true) ?>">
                &laquo;
        </a>
<?php else: ?>
        <a href="<?= $this->url(null, array('page' => 1), true) ?>" class="disable\
d">
                &laquo;
        </a>
<?php endif;?>

<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
        <a href="<?= $this->url(null, array('page' => $this->previous), true) ?>">
                &lt;
        </a>
<?php else: ?>
        <a href="#" class="disabled">&lt;</a>
<?php endif; ?>

<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
        <?php if ($page != $this->current): ?>
                <a href="<?= $this->url(null, array('page' => $page), true) ?>" class="bt\
n">
                        <?php echo $page; ?>
                </a>
        <?php else: ?>
                <a href="#" class="btn disabled"><?php echo $page; ?></a>
        <?php endif; ?>
<?php endforeach; ?>
<!-- Next page link -->

<?php if (isset($this->next)): ?>
        <a href="<?= $this->url(null, array('page' => $this->next), true) ?>">
                &gt;
        </a>
<?php else: ?>
        <a href="#" class="disabled">&gt;</a>
<?php endif; ?>

<?php if($this->current != $this->pageCount): ?>
        <a href="<?= $this->url(null, array('page' => $this->pageCount), true) ?>"\
```

```
46  >
47                  &raquo;
48          </a>
49  <?php else: ?>
50          <a href="<?= $this->url(null, array('page' => 1), true) ?>" class="disable\
51  d">
52                  &raquo;
53          </a>
54  <?php endif;?>
55
56  </ul>
57  <span class="info">
58      <?php echo $this->current;?> de
59      <?php echo $this->pageCount;?>
60  </span>
61  </div>
62  <?php endif; ?>
```

https://gist.github.com/4011827

Ele é baseado em um dos exemplos que a Zend fornece no manual do framework

Podemos agora executar os testes do controlador e verificar se está tudo funcionando:

```
1  phpunit -c module/Application/tests/phpunit.xml --group=Controller
```

Módulos

Nosso projeto de blog precisa agora de outra funcionalidade importante: a criação dos posts. Mas não queremos que qualquer visitante possa alterar ou remover posts, apenas os usuários com permissão para tal. Uma forma de fazermos isso e mantermos o código organizado é criando um novo módulo, o *Admin*.

O conceito de módulos já existia nas versões anteriores do framework, mas no Zend Framework 2 ele ganhou uma importância maior. Módulos são agrupamentos de códigos que podem ser facilmente incluídos em qualquer projeto, sendo totalmente desacoplados. Podemos criar aqui um módulo *Admin* totalmente genérico e usá-lo em qualquer projeto que criarmos no futuro, como um *CMS* ou um fórum.

Existem diversos módulos prontos, criados pela comunidade de desenvolvedores e distribuídos gratuitamente no site http://modules.zendframework.com.

Vamos criar o nosso módulo *Admin* usando como base o módulo *Skel*, para agilizar o processo.

O primeiro passo é duplicar o diretório *module/Skel* criando o diretório *module/Admin*. No *Linux/Mac*:

```
1 cp -r module/Skel/ module/Admin
```

Configurando o novo módulo

Passos:

Alterar o *namespace* no arquivo *module/Admin/Module.php*:

```
1 namespace Admin;
```

Alterar o *module/Admin/config/module.config.php*

```
1  <?php
2
3  return array(
4      'controllers' => array( //add module controllers
5          'invokables' => array(
6              'Admin\Controller\Index' => 'Admin\Controller\IndexController',
7          ),
8      ),
9
10     'router' => array(
11         'routes' => array(
12             'admin' => array(
13                 'type'    => 'Literal',
14                 'options' => array(
15                     'route'    => '/admin',
16                     'defaults' => array(
17                         '__NAMESPACE__' => 'Admin\Controller',
18                         'controller'    => 'Index',
19                         'action'        => 'index',
```

```
20                          'module'        => 'admin'
21                      ),
22                  ),
23                  'may_terminate' => true,
24                  'child_routes' => array(
25                      'default' => array(
26                          'type'    => 'Segment',
27                          'options' => array(
28                              'route'    => '/[:controller[/:action]]',
29                              'constraints' => array(
30                                  'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
31                                      'action'     => '[a-zA-Z][a-zA-Z0-9_-]*\
32 ',
33                              ),
34                          'defaults' => array(),
35                      ),
36                      //permite mandar dados pela url
37                      'child_routes' => array(
38                          'wildcard' => array(
39                              'type' => 'Wildcard'
40                          ),
41                      ),
42                  ),
43              ),
44          ),
45      ),
46  ),
47  'view_manager' => array(
48          //the module can have a specific layout
49          /* 'template_map' => array(
50                 'layout/layout' => __DIR__ . '/../view/layout/layout.phtml',
51           ),*/
52          'template_path_stack' => array(
53              'admin' => __DIR__ . '/../view',
54          ),
55      ),
56      //module can have a specific db configuration
57      /* 'db' => array(
58            'driver' => 'PDO_SQLite',
59            'dsn' => 'sqlite:' . __DIR__ .'/../data/admin.db',
60            'driver_options' => array(
61                PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
62            )
63      )*/
64  );
```

https://gist.github.com/4012030

O arquivo *module.config.php* é o responsável por configurar todo o comportamento do módulo.

No Zend Framework 1 existia muita "mágica", muito comportamento que era automaticamente executado. Isso acarretava menos código para o desenvolvedor, mas retirava um pouco da performance e flexibilidade. Com o Zend Framework 2 as coisas são bem mais explícitas, praticamente todo o comportamento é passivel de alteração e modificação. Isso aumentou o número e a complexidade dos arquivos de configuração mas deu um grande aumento de performance e flexibilidade.

O *module.config.php* é um bom exemplo dsso. Como podemos ver no arquivo acima praticamente tudo é configurado, desde quais são os controladores disponíveis (*array controllers*) até a rota (*array router*). Isso nos trouxe a flexibilidade de podermos ter um *layout* e visões específicas para o módulo (*array view_manager*) e até uma conexão diferente com o banco de dados (*array db*).

Voltaremos a alterar o *module.config.php* nos próximos tópicos, para incluir um novo controlador por exemplo. Mas alguns pontos dificilmente precisaremos alterar, notadamente o *array router* pois a configuração é genérica o suficiente para a grande maioria dos casos.

Configurar diretórios

Podemos remover o diretório *Skel* dentro do *src* e criar um novo diretório chamado *src/Admin* com seus sub-diretórios:

```
rm -rf module/Admin/src/Skel
mkdir module/Admin/src/Admin
mkdir module/Admin/src/Admin/Controller
mkdir module/Admin/src/Admin/Model
mkdir module/Admin/src/Admin/Service
```

Faremos o mesmo com o diretório *tests*:

```
rm -rf module/Admin/tests/src/Skel
mkdir module/Admin/tests/src/Admin
mkdir module/Admin/tests/src/Admin/Controller
mkdir module/Admin/tests/src/Admin/Model
mkdir module/Admin/tests/src/Admin/Service
```

Conforme comentado no primeiro capítulo, o framework ainda não possui uma ferramenta para facilitar a criação desta estrutura de diretórios e no momento o indicado é usarmos estes projetos *Skel* como modelos.

Alterar as configurações do *tests/Bootstrap.php*

```
static function getModulePath()
{
        //mudar o caminho do modulo
        return __DIR__ . '/../../../module/Admin';
}
```

Adicionar o módulo no *application.config.php*

Para cada módulo que criarmos no projeto sempre precisaremos incluí-lo no *application.config.php* para que ele esteja disponível aos usuários:

```
'modules' => array(
        'Application',
        'Core',
        //'Skel',
        'Admin'
),
```

Temos agora três módulos no sistema, dois deles independentes entre si (o *Application* e o *Admin*), sendo que ambos dependem apenas do módulo *Core*.

Configurar os dados para os testes

Precisamos agora configurar os dados para os testes do módulo *Admin*. O arquivo *module/Admin/data/-test.data.php* vai ter os comandos *SQL* do arquivo do *Application* mais o comando de criação da tabela *user*. Essa duplicação de informação pode ser resolvida com mudanças na estrutura dos testes, mas eu deixei desta forma para ser mais didático e não aumentar a complexidade. Recomendo a leitura avançada da documentação do *PHPUnit* antes de implementar essa estrutura em projetos maiores, principalmente a documentação sobre *Fixtures* e *Mocks*.

O conteúdo do arquivo *module/Admin/data/test.data.php*:

```
<?php
//queries used by tests
return array(
        'posts' => array(
                'create' => 'CREATE TABLE if not exists posts (
                        id INT NOT NULL AUTO_INCREMENT ,
                        title VARCHAR(250) NOT NULL ,
                        description TEXT NOT NULL ,
                        post_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
                        PRIMARY KEY (id) )
                        ENGINE = InnoDB;',
                'drop' => "DROP TABLE posts;"
        ),
        'comments' => array(
                'create' => 'CREATE TABLE if not exists comments (
                        id INT NOT NULL AUTO_INCREMENT ,
                         post_id INT NOT NULL ,
                        description TEXT NOT NULL ,
                        name VARCHAR(200) NOT NULL ,
                        email VARCHAR(250) NOT NULL ,
                        webpage VARCHAR(200) NOT NULL ,
                        comment_date TIMESTAMP NULL ,
                        PRIMARY KEY (id, post_id) ,
                        INDEX fk_comments_posts (post_id ASC) ,
                        CONSTRAINT fk_comments_posts
                        FOREIGN KEY (post_id )
                        REFERENCES posts (id )
                        ON DELETE NO ACTION
```

```
                        ON UPDATE NO ACTION)
                        ENGINE = InnoDB;',
                'drop' =>'drop table comments;'
        ),
        'users' => array(
                'create' => 'CREATE TABLE if not exists users (
                        id INT NOT NULL AUTO_INCREMENT ,
                        username VARCHAR(200) NOT NULL ,
                        password VARCHAR(250) NOT NULL ,
                        name VARCHAR(200) NULL ,
                        valid TINYINT NULL ,
                        role VARCHAR(20) NULL ,
                        PRIMARY KEY (id) )
                        ENGINE = InnoDB;' ,
                'drop'   => 'drop table users;',
        ),
);
```

https://gist.github.com/4012031

Criando a entidade User

Vamos agora criar o teste e o código da entidade *User* que vai ser usada pelo novo módulo.

Criamos o arquivo *module/Admin/tests/src/Admin/Model/UserTest.php*:

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

use Core\Test\ModelTestCase;
use Admin\Model\User;
use Zend\InputFilter\InputFilterInterface;

/**
 * @group Model
 */
class UserTest extends ModelTestCase
{
    public function testGetInputFilter()
    {
        $user = new User();
        $if = $user->getInputFilter();
        //testa se existem filtros
        $this->assertInstanceOf("Zend\InputFilter\InputFilter", $if);
        return $if;
    }

    /**
```

```
23      * @depends testGetInputFilter
24      */
25      public function testInputFilterValid($if)
26      {
27          //testa os filtros
28          $this->assertEquals(6, $if->count());
29          $this->assertTrue(
30              $if->has('id')
31          );
32          $this->assertTrue(
33              $if->has('username')
34          );
35          $this->assertTrue(
36              $if->has('password')
37          );
38          $this->assertTrue(
39              $if->has('name')
40          );
41          $this->assertTrue(
42              $if->has('valid')
43          );
44          $this->assertTrue(
45              $if->has('role')
46          );
47      }
48
49      /**
50      * @expectedException Core\Model\EntityException
51      */
52      public function testInputFilterInvalidoUsername()
53      {
54          //testa se os filtros estao funcionando
55          $user = new User();
56          //username so pode ter 50 caracteres
57          $user->username = 'Lorem Ipsum e simplesmente uma simulacao de text\
58  o da industria tipografica e de impressos. Lorem Ipsum e simplesmente uma s\
59  imulacao de texto da indústria tipografica e de impressos';
60      }
61
62      /**
63      * @expectedException Core\Model\EntityException
64      */
65      public function testInputFilterInvalidoRole()
66      {
67          //testa se os filtros estao funcionando
68          $user = new User();
69          //role só pode ter 20 caracteres
70          $user->role = 'Lorem Ipsum e simplesmente uma simulacao de texto da\
71   industria tipografica e de impressos. Lorem Ipsum e simplesmente uma simul\
```

```
72  acao de texto da indústria tipografica e de impressos';
73      }
74
75      /**
76      * Teste de insercao de um user valido
77      */
78      public function testInsert()
79      {
80          $user = $this->addUser();
81          //testa o filtro de tags e espaços
82          $this->assertEquals('Steve Jobs', $user->name);
83          //testa o auto increment da chave primaria
84          $this->assertEquals(1, $user->id);
85      }
86
87      /**
88      * @expectedException Core\Model\EntityException
89      * @expectedExceptionMessage Input inválido: username =
90      */
91      public function testInsertInvalido()
92      {
93          $user = new user();
94          $user->name = 'teste';
95          $user->username = '';
96
97          $saved = $this->getTable('Admin\Model\user')->save($user);
98      }
99
100     public function testUpdate()
101     {
102         $tableGateway = $this->getTable('Admin\Model\User');
103         $user = $this->addUser();
104
105         $id = $user->id;
106
107         $this->assertEquals(1, $id);
108
109         $user = $tableGateway->get($id);
110         $this->assertEquals('Steve Jobs', $user->name);
111
112         $user->name = 'Bill <br>Gates';
113         $updated = $tableGateway->save($user);
114
115         $user = $tableGateway->get($id);
116         $this->assertEquals('Bill Gates', $user->name);
117     }
118
119     /**
120     * @expectedException Core\Model\EntityException
```

```
121     * @expectedExceptionMessage Could not find row 1
122     */
123     public function testDelete()
124     {
125         $tableGateway = $this->getTable('Admin\Model\User');
126         $user = $this->addUser();
127         $id = $user->id;
128         $deleted = $tableGateway->delete($id);
129         $this->assertEquals(1, $deleted); //numero de linhas excluidas
130         $user = $tableGateway->get($id);
131         }
132 
133     private function addUser()
134     {
135         $user = new User();
136         $user->username = 'steve';
137         $user->password = md5('apple');
138         $user->name = 'Steve <b>Jobs</b>';
139         $user->valid = 1;
140         $user->role = 'admin';
141 
142         $saved = $this->getTable('Admin\Model\User')->save($user);
143         return $saved;
144     }
145 }
```

https://gist.github.com/4012034

O código é similar aos testes das entidades anteriores.

Vamos criar agora o código da entidade, no arquivo *module/Admin/src/Admin/Model/User.php*:

```
1  <?php
2  namespace Admin\Model;
3  
4  use Zend\InputFilter\Factory as InputFactory;
5  use Zend\InputFilter\InputFilter;
6  use Zend\InputFilter\InputFilterAwareInterface;
7  use Zend\InputFilter\InputFilterInterface;
8  use Core\Model\Entity;
9  
10 /**
11  * Entidade User
12  *
13  * @category Admin
14  * @package Model
15  */
16 class User extends Entity
17 {
18     /**
```

```
19      * Nome da tabela. Campo obrigatório
20      * @var string
21      */
22      protected $tableName ='users';
23
24      /**
25      * @var int
26      */
27      protected $id;
28
29      /**
30      * @var string
31      */
32      protected $username;
33
34      /**
35       * @var string
36       */
37      protected $password;
38
39      /**
40       * @var string
41       */
42      protected $name;
43
44      /**
45       * @var int
46       */
47      protected $valid;
48
49      /**
50       * @var string
51       */
52      protected $role;
53
54      /**
55       * Configura os filtros dos campos da entidade
56       *
57       * @return Zend\InputFilter\InputFilter
58       */
59      public function getInputFilter()
60      {
61          if (!$this->inputFilter) {
62              $inputFilter = new InputFilter();
63              $factory     = new InputFactory();
64
65              $inputFilter->add($factory->createInput(array(
66                  'name'     => 'id',
67                  'required' => true,
```

```
68                  'filters'  => array(
69                      array('name' => 'Int'),
70                  ),
71              )));
72
73              $inputFilter->add($factory->createInput(array(
74                  'name'     => 'username',
75                  'required' => true,
76                  'filters'  => array(
77                      array('name' => 'StripTags'),
78                      array('name' => 'StringTrim'),
79                  ),
80                  'validators' => array(
81                      array(
82                          'name'    => 'StringLength',
83                          'options' => array(
84                              'encoding' => 'UTF-8',
85                              'min'      => 1,
86                              'max'      => 50,
87                          ),
88                      ),
89                  ),
90              )));
91
92              $inputFilter->add($factory->createInput(array(
93                  'name'     => 'password',
94                  'required' => true,
95                  'filters'  => array(
96                      array('name' => 'StripTags'),
97                      array('name' => 'StringTrim'),
98                  ),
99              )));
100
101             $inputFilter->add($factory->createInput(array(
102                 'name'     => 'name',
103                 'required' => true,
104                 'filters'  => array(
105                     array('name' => 'StripTags'),
106                     array('name' => 'StringTrim'),
107                 ),
108             )));
109
110             $inputFilter->add($factory->createInput(array(
111                 'name'     => 'valid',
112                 'required' => true,
113                 'filters'  => array(
114                     array('name' => 'Int'),
115                 ),
116             )));
```

```
117
118             $inputFilter->add($factory->createInput(array(
119                 'name'     => 'role',
120                 'required' => true,
121                 'filters'  => array(
122                     array('name' => 'StripTags'),
123                     array('name' => 'StringTrim'),
124                 ),
125                 'validators' => array(
126                     array(
127                         'name'    => 'StringLength',
128                         'options' => array(
129                             'encoding' => 'UTF-8',
130                             'min'      => 1,
131                             'max'      => 20,
132                         ),
133                     ),
134                 ),
135             )));
136
137             $this->inputFilter = $inputFilter;
138         }
139
140         return $this->inputFilter;
141     }
142 }
```

https://gist.github.com/4001782

Podemos agora executar o testes do grupo *Model*:

```
1 phpunit -c module/Admin/tests/phpunit.xml  --group=Model
```

Os testes devem passar. Caso algum falhe é preciso revisar os códigos e testes.

Vamos agora trabalhar na próxima funcionalidade importante, a autenticação.

Serviços

Agora que temos um módulo administrativo precisamos identificar alguns usuários como membros válidos da administração do nosso blog, ou seja, precisamos de alguma forma de autenticação. Para isso vamos usar dois novos conceitos: o componente *Zend\Authentication* e os serviços.

O *Zend\Authentication* é um componente usado para autenticar os usuários de forma fácil e flexível. Podemos armazenar as credenciais dos usuários em vários modos, como banco de dados, *LDAP*, *HTTP*, etc, bastando escolher o *adapter* mais adequado. Ele também nos ajuda a manter as credenciais do usuário entre as páginas da nossa aplicação, o que vai ser muito útil para nós.

Quanto ao conceito de serviços. Nosso projeto atual possui apenas uma interface de gerenciamento *web* com a qual os usuários podem fazer a autenticação e postar novos textos. Poderíamos então colocar a lógica da autenticação em um controlador. Mas podemos no futuro precisar de uma outra interface, *mobile* por exemplo, e iremos precisar da mesma lógica de autenticação. A melhor forma de fazermos isso é criarmos um serviço, uma porção de código que pode ser facilmente usado por um controlador, outro serviço ou, mais tarde, uma *API*. O Zend Framework 2 usa intensamente o conceito de serviços e facilita a criação dos mesmos.

Serviço de autenticação

Vamos então criar um serviço chamado *Auth* que vai ser responsável pela autenticação dos usuários e, mais tarde, a autorização, verificando se determinado usuário pode ou não acessar uma *action*.

Iniciamos pelos testes, no arquivo *module/Admin/tests/src/Admin/Service/AuthTest.php*:

```
1  <?php
2  namespace Admin\Service;
3
4  use DateTime;
5  use Core\Test\ServiceTestCase;
6  use Admin\Model\User;
7  use Core\Model\EntityException;
8  use Zend\Authentication\AuthenticationService;
9
10 /**
11  * Testes do serviço Auth
12  * @category Admin
13  * @package Service
14  * @author  Elton Minetto<eminetto@coderockr.com>
15  */
16
17 /**
18  * @group Service
19  */
20 class AuthTest extends ServiceTestCase
21 {
22
23         /**
```

```
24          * Authenticação sem parâmetros
25          * @expectedException \Exception
26          * @return void
27          */
28          public function testAuthenticateWithoutParams()
29          {
30                  $authService = $this->serviceManager->get('Admin\Service\Auth');
31
32                  $authService->authenticate();
33          }
34
35          /**
36          * Authenticação sem parâmetros
37          * @expectedException \Exception
38          * @expectedExceptionMessage Parâmetros inválidos
39          * @return void
40                   */
41          public function testAuthenticateEmptyParams()
42          {
43                  $authService = $this->serviceManager->get('Admin\Service\Auth');
44                  $authService->authenticate(array());
45          }
46
47          /**
48          * Teste da autenticação inválida
49          * @expectedException \Exception
50          * @expectedExceptionMessage Login ou senha inválidos
51          * @return void
52          */
53          public function testAuthenticateInvalidParameters()
54          {
55                  $authService = $this->serviceManager->get('Admin\Service\Auth');
56                  $authService->authenticate(array(
57                          'username' => 'invalid', 'password' => 'invalid')
58                  );
59          }
60
61          /**
62          * Teste da autenticação Inválida
63          * @expectedException \Exception
64          * @expectedExceptionMessage Login ou senha inválidos
65          * @return void
66          */
67          public function testAuthenticateInvalidPassord()
68          {
69                  $authService = $this->serviceManager->get('Admin\Service\Auth');
70                  $user = $this->addUser();
71
72                  $authService->authenticate(array(
```

```
73                      'username' => $user->username, 'password' => 'invalida')
74              );
75      }
76
77      /**
78      * Teste da autenticação Válida
79      * @return void
80      */
81      public function testAuthenticateValidParams()
82      {
83              $authService = $this->serviceManager->get('Admin\Service\Auth');
84              $user = $this->addUser();
85
86              $result = $authService->authenticate(
87                      array('username' => $user->username, 'password' => 'apple')
88              );
89              $this->assertTrue($result);
90
91              //testar a se a authenticação foi criada
92              $auth = new AuthenticationService();
93              $this->assertEquals($auth->getIdentity(), $user->username);
94
95              //verica se o usuário foi salvo na sessão
96              $session = $this->serviceManager->get('Session');
97              $savedUser = $session->offsetGet('user');
98              $this->assertEquals($user->id, $savedUser->id);
99      }
100
101     /**
102     * Limpa a autenticação depois de cada teste
103     * @return void
104     */
105     public function tearDown()
106     {
107             parent::tearDown();
108             $auth = new AuthenticationService();
109             $auth->clearIdentity();
110     }
111
112     /**
113      Teste do logout
114     * @return void
115     */
116     public function testLogout()
117     {
118             $authService = $this->serviceManager->get('Admin\Service\Auth');
119             $user = $this->addUser();
120
121             $result = $authService->authenticate(
```

```
122                         array('username' => $user->username, 'password' => 'apple')
123                 );
124                 $this->assertTrue($result);
125 
126                 $result = $authService->logout();
127                 $this->assertTrue($result);
128 
129                 //verifica se removeu a identidade da autenticação
130                 $auth = new AuthenticationService();
131                 $this->assertNull($auth->getIdentity());
132 
133                 //verifica se o usuário foi removido da sessão
134                 $session = $this->serviceManager->get('Session');
135                 $savedUser = $session->offsetGet('user');
136                 $this->assertNull($savedUser);
137                 }
138 
139         private function addUser()
140         {
141                 $user = new User();
142                 $user->username = 'steve';
143                 $user->password = md5('apple');
144                 $user->name = 'Steve <b>Jobs</b>';
145                 $user->valid = 1;
146                 $user->role = 'admin';
147 
148                 $saved = $this->getTable('Admin\Model\User')->save($user);
149                 return $saved;
150         }
151 }
```

https://gist.github.com/4012038

Vamos analisar o que temos de novo no código acima.

```
1 $authService = $this->serviceManager->get('Admin\Service\Auth');
```

Estamos usando o componente *ServiceManager* do Zend Framework 2 para instanciar nosso serviço de autenticação (e a *Session*). Vou explicar sobre ele daqui a pouco, quando criarmos o código do serviço.

```
1 public function tearDown()
```

Essa é uma função do próprio *PHPUnit* que estamos estendendo aqui. Ela é executada sempre após todos os testes e nós estamos garantindo que a identidade do usuário seja removida para evitar problemas com outros testes.

Vamos agora criar o código do nosso serviço e explicar os novos conceitos envolvidos no código. O arquivo *module/Admin/src/Admin/Service/Auth.php*:

```php
<?php
namespace Admin\Service;

use Core\Service\Service;
use Zend\Authentication\AuthenticationService;
use Zend\Authentication\Adapter\DbTable as AuthAdapter;
use Zend\Db\Sql\Select;

/**
 * Serviço responsável pela autenticação da aplicação
*
* @category Admin
* @package Service
* @author  Elton Minetto<eminetto@coderockr.com>
*/
class Auth extends Service
{
        /**
        * Adapter usado para a autenticação
        * @var Zend\Db\Adapter\Adapter
        */
        private $dbAdapter;

        /**
        * Construtor da classe
        *
        * @return void
        */
        public function __construct($dbAdapter = null)
        {
                $this->dbAdapter = $dbAdapter;
        }

        /**
        * Faz a autenticação dos usuários
        *
        * @param array $params
        * @return array
        */
        public function authenticate($params)
        {
                if (!isset($params['username']) || !isset($params['password'])) {
                        throw new \Exception("Parâmetros inválidos");
                }

                $password = md5($params['password']);
                $auth = new AuthenticationService();
                $authAdapter = new AuthAdapter($this->dbAdapter);
                $authAdapter
```

```
50                          ->setTableName('users')
51                          ->setIdentityColumn('username')
52                          ->setCredentialColumn('password')
53                          ->setIdentity($params['username'])
54                          ->setCredential($password);
55                  $result = $auth->authenticate($authAdapter);
56
57                  if (! $result->isValid()) {
58                          throw new \Exception("Login ou senha inválidos");
59                  }
60
61                  //salva o user na sessão
62                  $session = $this->getServiceManager()->get('Session');
63                  $session->offsetSet('user', $authAdapter->getResultRowObject());
64
65                  return true;
66          }
67
68          /**
69          * Faz o logout do sistema
70          *
71          * @return void
72          */
73          public function logout() {
74                  $auth = new AuthenticationService();
75                  $session = $this->getServiceManager()->get('Session');
76                  $session->offsetUnset('user');
77                  $auth->clearIdentity();
78                  return true;
79          }
80  }
```

https://gist.github.com/4012039

Vamos aos detalhes:

```
1  class Auth extends Service
```

Para que uma classe seja considerada um serviço é preciso que ela implemente a interface *ServiceManagerAwareInterface*. A classe *Core\Service* faz essa implementação, por isso nossos serviços vão estendê-la. Outra vantagem dos serviços herdarem a mesma classe é a possibilidade que isso nos fornece de expansão de regras no futuro.

```
1  public function __construct($dbAdapter = null)
```

A instanciação do serviço precisa de uma conexão com o banco de dados, porque nesta versão iremos armazenar as credenciais dos usuários no banco de dados. Mais sobre isso nas próximas linhas.

```
1 public function authenticate($params)
```

Este é o método que faz a autenticação, recebendo um *array* com o *username* e o *password* do usuário para validação.

```
1 $password = md5($params['password']);
```

Vamos armazenar as senhas usando o padrão de *hash MD5*, por isso precisamos usar o mesmo método para podermos comparar a senha que o usuário digitou com a que está no banco de dados.

```
1 $auth = new AuthenticationService();
2 $authAdapter = new AuthAdapter($this->dbAdapter);
```

Criamos uma instância do componente *AuthenticationService* e indicamos qual é o adaptador onde estamos armazenando os dados do usuári (que iremos receber na instanciação do serviço).

```
1 $authAdapter
2         ->setTableName('users')
3         ->setIdentityColumn('username')
4         ->setCredentialColumn('password')
5         ->setIdentity($params['username'])
6         ->setCredential($password);
7 $result = $auth->authenticate($authAdapter);
```

Fazemos a autenticação, indicando qual é a tabela, a coluna do *username*, a coluna da senha e os dados que o usuário nos mandou. A variável *$result* possui o resultado da autenticação e podemos usá-lo para verificar se a mesma foi feita corretamente, o que fazemos nas próximas linhas do arquivo.

```
1 $session = $this->getServiceManager()->get('Session');
2 $session->offsetSet('user', $authAdapter->getResultRowObject());
```

Usamos aqui um novo componente, muito importante, a *Session*. Armazenar dados na sessão é algo que os desenvolvedores *PHP* vem fazendo a anos e imagino que seja um conceito já trivial. O Zend Framework implementa isso na forma do componente *Zend\Session\Container*, que possui os métodos *offsetSet* para armazenar conteúdos na sessão e *offsetGet* para recuperá-los.

ServiceManager

Vamos agora nos aprofundar um pouco nos conceitos do Zend Framework 2.

Como vimos no construtor da classe *Auth*, para instanciar o serviço precisamos passar como parâmetro uma instância de um *adapter* (no nosso caso uma conexão com o banco de dados) ou seja, a classe *Auth* tem uma dependência. A vantagem de usarmos isso é que o comportamento da classe fica mais flexível, pois podemos enviar uma conexão diferente dependendo do módulo, do ambiente de testes, etc.

O problema é que o processo de criação da instância de *Auth* ficou um pouco mais burocrático, pois precisamos criar primeiro o *adapter*. O Zend Framework fornece uma forma de nos auxiliar nesse processo, o *ServiceManager*. Podemos registrar nele todas as dependências das nossas classes e ele se encarrega de criá-las para nós, sempre que precisarmos.

Temos duas formas de configurá-lo. A primeira é criando um novo método no arquivo *module/Admin/-Module.php*:

```php
/**
 * Retorna a configuração do service manager do módulo
 * @return array
 */
public function getServiceConfig()
{
        return array(
                'factories' => array(
                        'Admin\Service\Auth' => function($sm) {
                                $dbAdapter = $sm->get('DbAdapter');
                                return new Service\Auth($dbAdapter);
                        },
                ),
        );
}
```

https://gist.github.com/4030845

A segunda é criando uma nova configuração no arquivo *module/Admin/config/module.config.php*:

```php
'service_manager' => array(
        'factories' => array(
                'Session' => function($sm) {
                        return new Zend\Session\Container('ZF2napratica');
                },
                'Admin\Service\Auth' => function($sm) {
                        $dbAdapter = $sm->get('DbAdapter');
                        return new Admin\Service\Auth($dbAdapter);
                },
        )
),
```

https://gist.github.com/4012041

Ambas as formas tem a mesma funcionalidade. Vamos usar a segunda delas, a alteração no *module.config.php*.

O que estamos dizendo ao framework é: "sempre que eu solicitar o serviço *Admin\Service\Auth* execute esta função anônima, use a conexão com o banco de dados atual e me retorne uma instância pronta do serviço". O mesmo para o serviço *Session*.

É por isso que no teste usamos:

```php
$authService = $this->serviceManager->get('Admin\Service\Auth');
```

Quando fazemos isso o *ServiceManager* se encarrega de criar uma nova instância, ou entregar uma que já exista e ele está armazenando para nós (podemos configurar o *ServiceManager* para sempre nos entregar uma nova instância, caso desejemos).

Perceba que dentro da função que cria a instância de *Auth* temos:

```
1 $dbAdapter = $sm->get('DbAdapter');
```

Estamos usando o *ServiceManager* para criar uma conexão com o banco de dados. A configuração do *DbAdapter* está no arquivo *module/Core/Module.php*.

O Zend Framework 2 usa o *ServiceManager* internamente para fazer a criação de praticamente tudo, desde controladores até visões. É um componente de grande utilidade.

Rodando os testes

Agora que criamos o código do serviço podemos rodar os testes novamente:

```
1 phpunit -c module/Admin/tests/phpunit.xml  --group=Service
```

Com o serviço de autenticação pronto podemos passar para o próximo tópico, que é usar o serviço na nossa aplicação.

Formulários

Agora que temos nosso serviço de autenticação precisamos criar um formulário para que o usuário possa fazer o *login* no sistema.

Formulário de login

O Zend Framework 2 possui um componente para auxiliar na criação dos formulários, o *Zend\Form*.

Vamos criar o nosso primeiro formulário, o *module/Admin/src/Admin/Form/Login.php*:

```
1  <?php
2  namespace Admin\Form;
3
4  use Zend\Form\Form;
5
6  class Login extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('login');
11         $this->setAttribute('method', 'post');
12         $this->setAttribute('action', '/admin/auth/login');
13
14         $this->add(array(
15             'name' => 'username',
16             'attributes' => array(
17                 'type'  => 'text',
18             ),
19             'options' => array(
20                 'label' => 'Username',
21             ),
22         ));
23         $this->add(array(
24             'name' => 'password',
25             'attributes' => array(
26                 'type'  => 'password',
27             ),
28             'options' => array(
29                 'label' => 'Password',
30             ),
31         ));
32         $this->add(array(
33             'name' => 'submit',
34             'attributes' => array(
35                 'type'  => 'submit',
36                 'value' => 'Entrar',
37                 'id' => 'submitbutton',
```

```
38              ),
39              ));
40      }
41  }
```

https://gist.github.com/4012044

O código é bastante explicativo, mas vou citar alguns pontos importantes:

```
1  $this->setAttribute('method', 'post');
2  $this->setAttribute('action', '/admin/auth/login');
```

Estamos configurando o método de envio dos dados para o método *POST* e indicando que a *ACTION* do formulário, ou seja, para onde os dados serão enviados, é a url */admin/auth/login*. Isso significa que iremos também criar um novo controlador neste capítulo, o *AuthController*.

```
1  $this->add(array(
2      'name' => 'username',
3      'attributes' => array(
4          'type'  => 'text',
5      ),
6      'options' => array(
7          'label' => 'Username',
8      ),
9  ));
```

Neste trecho estamos criando um *input HTML* do tipo *text* e configurando seu nome e *label*. Para cada *input* existe um componente no *namespace Zend\Form\Element*, como pode ser visto no manual do framework.

Controlador de autenticação

Vamos criar um novo controlador para fazer uso do nosso novo formulário e do serviço de autenticação que criamos no capítulo anterior.

Os testes para o novo controlador estão no *module/Admin/tests/src/Admin/Controller/AuthControllerTest.php*:

```php
<?php
use Core\Test\ControllerTestCase;
use Admin\Controller\AuthController;
use Admin\Model\User;
use Zend\Http\Request;
use Zend\Stdlib\Parameters;
use Zend\View\Renderer\PhpRenderer;

/**
* @group Controller
*/
class AuthControllerTest extends ControllerTestCase
{
    /**
    * Namespace completa do Controller
    * @var string
    */
    protected $controllerFQDN = 'Admin\Controller\AuthController';

    /**
    * Nome da rota. Geralmente o nome do módulo
    * @var string
    */
    protected $controllerRoute = 'admin';

    public function test404()
    {
        $this->routeMatch->setParam('action', 'action_nao_existente');
        $result = $this->controller->dispatch($this->request);
        $response = $this->controller->getResponse();
        $this->assertEquals(404, $response->getStatusCode());
    }

    public function testIndexActionLoginForm()
    {
        // Invoca a rota index
        $this->routeMatch->setParam('action', 'index');
        $result = $this->controller->dispatch(
            $this->request, $this->response
        );

        // Verifica o response
        $response = $this->controller->getResponse();
        $this->assertEquals(200, $response->getStatusCode());

        // Testa se um ViewModel foi retornado
        $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);

        // Testa os dados da view
```

```
50          $variables = $result->getVariables();
51
52          $this->assertArrayHasKey('form', $variables);
53
54          // Faz a comparação dos dados
55          $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
56          $form = $variables['form'];
57          //testa os ítens do formulário
58          $username = $form->get('username');
59          $this->assertEquals('username', $username->getName());
60          $this->assertEquals('text', $username->getAttribute('type'));
61
62          $password = $form->get('password');
63          $this->assertEquals('password', $password->getName());
64          $this->assertEquals('password', $password->getAttribute('type'));
65      }
66
67      /**
68      * @expectedException Exception
69      * @expectedExceptionMessage Acesso inválido
70      */
71      public function testLoginInvalidMethod()
72      {
73          $user = $this->addUser();
74
75          // Invoca a rota index
76          $this->routeMatch->setParam('action', 'login');
77          $result = $this->controller->dispatch(
78              $this->request, $this->response
79          );
80      }
81
82      public function testLogin()
83      {
84          $user = $this->addUser();
85
86          // Invoca a rota index
87          $this->request->setMethod('post');
88          $this->request->getPost()->set('username', $user->username);
89          $this->request->getPost()->set('password', 'apple');
90
91          $this->routeMatch->setParam('action', 'login');
92          $result = $this->controller->dispatch(
93              $this->request, $this->response
94          );
95
96          // Verifica o response
97          $response = $this->controller->getResponse();
98          //deve ter redirecionado
```

```
99          $this->assertEquals(302, $response->getStatusCode());
100         $headers = $response->getHeaders();
101         $this->assertEquals('Location: /', $headers->get('Location'));
102     }
103
104     public function testLogout()
105     {
106         $user = $this->addUser();
107
108         $this->routeMatch->setParam('action', 'logout');
109         $result = $this->controller->dispatch(
110             $this->request, $this->response
111         );
112
113         // Verifica o response
114         $response = $this->controller->getResponse();
115         //deve ter redirecionado
116         $this->assertEquals(302, $response->getStatusCode());
117         $headers = $response->getHeaders();
118         $this->assertEquals('Location: /', $headers->get('Location'));
119     }
120
121     private function addUser()
122     {
123         $user = new User();
124         $user->username = 'steve';
125         $user->password = md5('apple');
126         $user->name = 'Steve <b>Jobs</b>';
127         $user->valid = 1;
128         $user->role = 'admin';
129
130         $saved = $this->getTable('Admin\Model\User')->save($user);
131         return $saved;
132     }
133 }
```

https://gist.github.com/4012046

O teste é parecido com os anteriores. A principal diferença é testarmos a existência de uma instância de *Zend\Form\Form* e os seus elementos.

Vamos agora criar o código do novo controlador, no arquivo *module/Admin/src/Admin/Controller/AuthController.php*:

```php
<?php
namespace Admin\Controller;

use Zend\View\Model\ViewModel;
use Core\Controller\ActionController;
use Admin\Form\Login;

/**
* Controlador que gerencia os posts
*
* @category Admin
* @package Controller
* @author  Elton Minetto<eminetto@coderockr.com>
*/
class AuthController extends ActionController
{
    /**
    * Mostra o formulário de login
    * @return void
    */
    public function indexAction()
    {
        $form = new Login();
        return new ViewModel(array(
            'form' => $form
        ));
    }

    /**
    * Faz o login do usuário
    * @return void
    */
    public function loginAction()
    {
        $request = $this->getRequest();
        if (!$request->isPost()) {
            throw new \Exception('Acesso inválido');
        }

        $data = $request->getPost();
        $service = $this->getService('Admin\Service\Auth');
        $auth = $service->authenticate(
            array(
                'username' => $data['username'],
                'password' => $data['password']
            )
        );

        return $this->redirect()->toUrl('/');
```

```
50      }
51
52      /**
53      * Faz o logout do usuário
54      * @return void
55      */
56      public function logoutAction()
57      {
58          $service = $this->getService('Admin\Service\Auth');
59          $auth = $service->logout();
60
61          return $this->redirect()->toUrl('/');
62      }
63  }
```

https://gist.github.com/4012047

Alguns detalhes:

```
1  public function indexAction()
```

Instancia o *Form* e o envia para a visão. Nada de complexo nesse ponto.

```
1  public function loginAction()
2  public function logoutAction()
```

Usamos agora o nosso serviço de autenticação. Note que usamos o método *$this->getService()* que é uma facilidade da classe *Core/ActionController* que simplifica a chamada do *ServiceManager*, sem alterar em nada seu funcionamento.

Precisamos adicionar o novo controlador na configuração do nosso módulo. Para isso vamos adicionar uma nova linha no array *invokables* do arquivo *module/Admin/config/module.config.php*:

```
1  'Admin\Controller\Auth' => 'Admin\Controller\AuthController',
```

O último passo é criarmos a visão para o novo controlador. Precisamos criar o diretório:

```
1  mkdir module/Admin/view/admin/auth
```

E o arquivo *module/Admin/view/admin/auth/index.phtml*:

```
1  <?php
2  echo $this->form()->openTag($form);
3  echo $this->formCollection($form);
4  echo $this->form()->closeTag();
```

É o que precisamos para que o formulário seja transformado em *HTML*. Podemos mudar a forma como o código *HTML* é gerado ao alterar as configurações do objeto *form*. No manual do framework temos mais detalhes sobre isso.

Podemos agora executar os testes novamente:

```
phpunit -c module/Admin/tests/phpunit.xml  --group=Controller
```

CRUD de posts

Agora que temos o *login* do usuário funcionando podemos criar a próxima funcionalidade, que é a criação e alteração dos posts. Para isso criamos um novo formulário, mas vamos armazená-lo no módulo *Application*, por motivos de organização. Então o arquivo *module/Application/src/Application/Form/Post.php* ficou:

```
<?php
namespace Application\Form;

use Zend\Form\Form;

class Post extends Form
{
    public function __construct()
    {
        parent::__construct('post');
        $this->setAttribute('method', 'post');
        $this->setAttribute('action', '/admin/index/save');

        $this->add(array(
            'name' => 'id',
            'attributes' => array(
                'type'  => 'hidden',
            ),
        ));

        $this->add(array(
            'name' => 'title',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Título',
            ),
        ));
        $this->add(array(
            'name' => 'description',
            'attributes' => array(
                'type'  => 'textarea',
            ),
            'options' => array(
                'label' => 'Texto do post',
            ),
        ));
        $this->add(array(
```

```
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Enviar',
                'id' => 'submitbutton',
            ),
        ));
    }
}
```

https://gist.github.com/4012050

Como não temos nada de novo nesse código vamos passar para a criação do *module/Admin/tests/src/Admin/Controller/IndexControllerTest.php*:

```
<?php
use Core\Test\ControllerTestCase;
use Admin\Controller\IndexController;
use Application\Model\Post;
use Zend\Http\Request;
use Zend\Stdlib\Parameters;
use Zend\View\Renderer\PhpRenderer;

/**
 * @group Controller
 */
class IndexControllerTest extends ControllerTestCase
{
    /**
    * Namespace completa do Controller
    * @var string
    */
    protected $controllerFQDN = 'Admin\Controller\IndexController';

    /**
    * Nome da rota. Geralmente o nome do módulo
    * @var string
    */
    protected $controllerRoute = 'admin';

    /**
    * Testa o acesso a uma action que não existe
    */
    public function test404()
    {
        $this->routeMatch->setParam('action', 'action_nao_existente');
        $result = $this->controller->dispatch($this->request);
        $response = $this->controller->getResponse();
        $this->assertEquals(404, $response->getStatusCode());
    }
```

```
36
37      /**
38      * Testa a tela de inclusão de um novo registro
39      * @return void
40      */
41      public function testSaveActionNewRequest()
42      {
43          // Dispara a ação
44          $this->routeMatch->setParam('action', 'save');
45          $result = $this->controller->dispatch(
46              $this->request, $this->response
47          );
48          // Verifica a resposta
49          $response = $this->controller->getResponse();
50          $this->assertEquals(200, $response->getStatusCode());
51
52          // Testa se recebeu um ViewModel
53          $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
54
55          //verifica se existe um form
56          $variables = $result->getVariables();
57          $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
58          $form = $variables['form'];
59          //testa os ítens do formulário
60          $id = $form->get('id');
61          $this->assertEquals('id', $id->getName());
62          $this->assertEquals('hidden', $id->getAttribute('type'));
63      }
64
65      /**
66      * Testa a alteração de um post
67      */
68      public function testSaveActionUpdateFormRequest()
69      {
70          $postA = $this->addPost();
71
72          // Dispara a ação
73          $this->routeMatch->setParam('action', 'save');
74          $this->routeMatch->setParam('id', $postA->id);
75          $result = $this->controller->dispatch(
76              $this->request, $this->response
77          );
78
79          // Verifica a resposta
80          $response = $this->controller->getResponse();
81          $this->assertEquals(200, $response->getStatusCode());
82
83          // Testa se recebeu um ViewModel
84          $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
```

```
85          $variables = $result->getVariables();
86
87          //verifica se existe um form
88          $variables = $result->getVariables();
89          $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
90          $form = $variables['form'];
91
92          //testa os ítens do formulário
93          $id = $form->get('id');
94          $title = $form->get('title');
95          $this->assertEquals('id', $id->getName());
96          $this->assertEquals($postA->id, $id->getValue());
97          $this->assertEquals($postA->title, $title->getValue());
98      }
99
100     /**
101     * Testa a inclusão de um novo post
102     */
103     public function testSaveActionPostRequest()
104     {
105         // Dispara a ação
106         $this->routeMatch->setParam('action', 'save');
107
108         $this->request->setMethod('post');
109         $this->request->getPost()->set('title', 'Apple compra a Coderockr')\
110 ;
111         $this->request->getPost()->set(
112             'description', 'A Apple compra a <b>Coderockr</b><br> '
113         );
114
115         $result = $this->controller->dispatch(
116             $this->request, $this->response
117         );
118         // Verifica a resposta
119         $response = $this->controller->getResponse();
120         //a página redireciona, então o status = 302
121         $this->assertEquals(302, $response->getStatusCode());
122
123         //verifica se salvou
124         $posts = $this->getTable('Application\Model\Post')
125                       ->fetchAll()
126                       ->toArray();
127         $this->assertEquals(1, count($posts));
128         $this->assertEquals(
129             'Apple compra a Coderockr', $posts[0]['title']
130         );
131         $this->assertNotNull($posts[0]['post_date']);
132     }
133
```

```
134     /**
135     * Tenta salvar com dados inválidos
136     *
137     */
138     public function testSaveActionInvalidPostRequest()
139     {
140         // Dispara a ação
141         $this->routeMatch->setParam('action', 'save');
142 
143         $this->request->setMethod('post');
144         $this->request->getPost()->set('title', '');
145 
146         $result = $this->controller->dispatch(
147             $this->request, $this->response
148         );
149 
150         //verifica se existe um form
151         $variables = $result->getVariables();
152         $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
153         $form = $variables['form'];
154 
155         //testa os errors do formulário
156         $title = $form->get('title');
157         $titleErrors = $title->getMessages();
158         $this->assertEquals(
159             "Value is required and can't be empty",
160             $titleErrors['isEmpty']
161         );
162 
163         $description = $form->get('description');
164         $descriptionErrors = $description->getMessages();
165         $this->assertEquals(
166             "Value is required and can't be empty",
167             $descriptionErrors['isEmpty']
168         );
169     }
170 
171     /**
172     * Testa a exclusão sem passar o id do post
173     * @expectedException Exception
174     * @expectedExceptionMessage Código obrigatório
175     */
176     public function testInvalidDeleteAction()
177     {
178         $post = $this->addPost();
179         // Dispara a ação
180         $this->routeMatch->setParam('action', 'delete');
181         $result = $this->controller->dispatch(
182             $this->request, $this->response
```

```
183         );
184         // Verifica a resposta
185         $response = $this->controller->getResponse();
186     }
187 
188     /**
189     * Testa a exclusão do post
190     */
191     public function testDeleteAction()
192     {
193         $post = $this->addPost();
194         // Dispara a ação
195         $this->routeMatch->setParam('action', 'delete');
196         $this->routeMatch->setParam('id', $post->id);
197 
198         $result = $this->controller->dispatch(
199             $this->request, $this->response
200         );
201         // Verifica a resposta
202         $response = $this->controller->getResponse();
203         //a página redireciona, então o status = 302
204         $this->assertEquals(302, $response->getStatusCode());
205 
206         //verifica se excluiu
207         $posts = $this->getTable('Application\Model\Post')
208                       ->fetchAll()
209                       ->toArray();
210         $this->assertEquals(0, count($posts));
211     }
212 
213     /**
214     * Adiciona um post para os testes
215     */
216     private function addPost()
217     {
218         $post = new Post();
219         $post->title = 'Apple compra a Coderockr';
220         $post->description = 'A Apple compra a <b>Coderockr</b><br> ';
221         $post->post_date = date('Y-m-d H:i:s');
222         $saved = $this->getTable('Application\Model\Post')->save($post);
223         return $saved;
224     }
225 }
```

https://gist.github.com/4012052

Além de testarmos a existência de um formulário e seus elementos, agora realizamos todos os testes para a inclusão, alteração e remoção de um post. Esse é o maior arquivo de testes que temos no projeto mas imagino que com o que aprendemos até o momento será fácil entender o código e os comentários.

O código do *module/Admin/src/Admin/Controller/IndexController.php*:

```php
<?php
namespace Admin\Controller;

use Zend\View\Model\ViewModel;
use Core\Controller\ActionController;
use Application\Model\Post;
use Application\Form\Post as PostForm;

/**
* Controlador que gerencia os posts
*
* @category Admin
* @package Controller
* @author  Elton Minetto <eminetto@coderockr.com>
*/
class IndexController extends ActionController
{
    /**
    * Cria ou edita um post
    * @return void
    */
    public function saveAction()
    {
        $form = new PostForm();
        $request = $this->getRequest();
        /* se a requisição é post os dados foram enviados via formulário*/
        if ($request->isPost()) {
            $post = new Post;
            /* configura a validação do formulário com os filtros
             e validators da entidade*/
            $form->setInputFilter($post->getInputFilter());
            /* preenche o formulário com os dados que o usuário digitou na \
tela*/
            $form->setData($request->getPost());
            /* faz a validação do formulário*/
            if ($form->isValid()) {
                /* pega os dados validados e filtrados */
                $data = $form->getData();
                /* remove o botão submit dos dados pois ele não vai
                ser salvo na tabela*/
                unset($data['submit']);
                /* armazena a data de inclusão do post*/
                $data['post_date'] = date('Y-m-d H:i:s');
                /* preenche os dados do objeto Post com os dados do formulá\
rio*/
                $post->setData($data);
                /* salva o novo post*/
                $saved =  $this->getTable('Application\Model\Post')->save($\
post);
```

```
50                  /* redireciona para a página inicial*/
51                  return $this->redirect()->toUrl('/');
52                      }
53              }
54          /* essa é a forma de recuperar um parâmetro vindo da url como:
55              http://zfnapratica.dev/admin/index/save/id/1
56          */
57          $id = (int) $this->params()->fromRoute('id', 0);
58          if ($id > 0) {
59              /* busca a entidade no banco de dados*/
60              $post = $this->getTable('Application\Model\Post')->get($id);
61              /* preenche o formulário com os  dados do banco de dados*/
62              $form->bind($post);
63              /* muda o texto do botão submit*/
64              $form->get('submit')->setAttribute('value', 'Edit');
65          }
66          return new ViewModel(
67              array('form' => $form)
68          );
69      }
70
71      /**
72      * Exclui um post
73      * @return void
74      */
75      public function deleteAction()
76      {
77          $id = (int) $this->params()->fromRoute('id', 0);
78          if ($id == 0) {
79              throw new \Exception("Código obrigatório");
80          }
81              /* remove o registro e redireciona para a página inicial*/
82          $this->getTable('Application\Model\Post')->delete($id);
83          return $this->redirect()->toUrl('/');
84      }
85  }
```

https://gist.github.com/4012053

Esse controlador inclui diversos novos e importantes conceitos. Para facilitar o entendimento de cada um deles eu inclui os comentários explicando cada trecho de código. Recomendo a leitura com atenção do código acima.

Um dos trechos mais interessantes é o *$form->setInputFilter($post->getInputFilter());* pois usamos a configuração de validação da entidade direto no formulário, sem precisar reescrever nada. Isso ajuda muito na manutenção das regras de validação.

Criaremos agora a visão para nosso novo controlador, no arquivo *module/Admin/view/admin/index/save.phtml*:

```php
<?php
echo $this->form()->openTag($form);
echo $this->formCollection($form);
echo $this->form()->closeTag();
```

O último passo é habilitar o novo controlador no *module/Admin/config/module.config.php*:

```
'controllers' => array( //add module controllers
        'invokables' => array(
                'Admin\Controller\Index' => 'Admin\Controller\IndexController',
                'Admin\Controller\Auth' => 'Admin\Controller\AuthController',
        ),
),
```

Rodando os testes podemos verificar se o novo controlador está correto:

```
phpunit -c module/Admin/tests/phpunit.xml
```

Eventos

Incluindo a autenticação

Adicionaremos uma nova funcionalidade no quesito segurança.

A idéia de termos um módulo de administração é permitir que somente usuário autenticados possam acessar determinadas ações, como remover um post. Fizemos a autenticação do usuário no capítulo anterior, mas não fizemos nada para garantir que somente usuários autenticados possam acessar o controlador *Admin\IndexController*.

Vamos fazer isso usando um conceito bem interessante do Zend Framework 2, os eventos. Os eventos são gerenciados por um componente chamado *EventManager* que usaremos nesse capítulo.

Praticamente todos os componentes do Zend Framework executam eventos que podemos interceptar e colocar nossa lógica para responder a este evento. Vamos usar um evento chamado *EVENT_DISPATCH* que é gerado pela classe *Zend\Mvc\Controller\AbstractActionController* (e suas classes filhas) toda vez que um controlador é executado.

Para fazer isso vamos adicionar um código que vai "ouvir" (o termo usado pelo framework é *listener* ou "ouvinte") este evento e executar um novo método do serviço de autenticação. Vamos adicionar dois métodos no arquivo *module/Admin/Module.php*:

```
1  /**
2   * Executada no bootstrap do módulo
3   *
4   * @param MvcEvent $e
5   */
6  public function onBootstrap($e)
7  {
8          $moduleManager = $e->getApplication()
9                            ->getServiceManager()
10                           ->get('modulemanager');
11         $sharedEvents = $moduleManager->getEventManager()
12                                       ->getSharedManager();
13
14         //adiciona eventos ao módulo
15         $sharedEvents->attach(
16             'Zend\Mvc\Controller\AbstractActionController',
17             MvcEvent::EVENT_DISPATCH,
18             array($this, 'mvcPreDispatch'),
19             100
20         );
21 }
22
23 /**
24 * Verifica se precisa fazer a autorização do acesso
25 * @param  MvcEvent $event Evento
26 * @return boolean
27 */
28 public function mvcPreDispatch($event)
```

```
29 {
30         $di = $event->getTarget()->getServiceLocator();
31         $routeMatch = $event->getRouteMatch();
32         $moduleName = $routeMatch->getParam('module');
33         $controllerName = $routeMatch->getParam('controller');
34 
35         if (
36             $moduleName == 'admin' &&
37             $controllerName != 'Admin\Controller\Auth'
38         ) {
39                 $authService = $di->get('Admin\Service\Auth');
40                 if (! $authService->authorize()) {
41                         $redirect = $event->getTarget()->redirect();
42                         $redirect->toUrl('/admin/auth');
43                 }
44         }
45         return true;
46 }
```

https://gist.github.com/4012056

Usamos o método *public function onBootstrap($e)* que é executado na inicialização do módulo para incluir um novo *listener* para o evento:

```
1 $sharedEvents->attach(
2     'Zend\Mvc\Controller\AbstractActionController',
3     MvcEvent::EVENT_DISPATCH,
4     array($this, 'mvcPreDispatch'),
5     100
6  );
```

Basicamente o comando acima diz para o *EventManager*: "sempre que o *AbstractActionController* gerar o evento *EVENT_DISPATCH* execute o código do método *mvcPreDispatch* desta classe, com a prioridade 100". O parâmetro da prioridade nos ajuda a ter vários *listeners* ouvindo o mesmo evento e sendo executados de acordo com a sua prioridade (os maiores serão executados primeiro).

O método *mvcPreDispatch* recebe como parâmetro o evento que foi gerado, pega as informações da requisição (controlador e módulo) e verifica se deve ou não chamar o método *authorize* do serviço de autenticação. Caso a autorização retorne *false* o usuário é redirecionado para a página inicial.

Vamos adicionar o teste para o novo método no nosso *AuthTest.php*:

```php
/**
 * Teste da autorização
 * @return void
*/
public function testAuthorize()
{
        $authService = $this->getService('Admin\Service\Auth');

        $result = $authService->authorize();
        $this->assertFalse($result);

        $user = $this->addUser();

        $result = $authService->authenticate(
                array('username' => $user->username, 'password' => 'apple')
        );
        $this->assertTrue($result);

        $result = $authService->authorize();
        $this->assertTrue($result);
}
```

https://gist.github.com/4012057

Nada de novo, é um teste bem simples. Adicionaremos mais testes no próximo capítulo.

Vamos incluir o novo método no serviço *Auth*:

```php
/**
* Faz a autorização do usuário para acessar o recurso
* @return boolean
*/
public function authorize()
{
        $auth = new AuthenticationService();
        if ($auth->hasIdentity()) {
                return true;
        }
        return false;
}
```

https://gist.github.com/4012058

O que este método faz é basicamente verificar se algum usuário fez a autenticação, e caso não tenha feito retornar falso.

Agora podemos executar novamente os testes, conforme os exemplos dos capítulos anteriores.

Neste exemplo nós criamos um código para ouvir um evento que o framework gerou. Mas podemos criar eventos em nossas classes e criar códigos para interceptá-los, o que cria uma grande quantidade de possibilidades para nossos projetos. Mais detalhes e exemplos podem ser encontrados no manual.

Controle de Acesso

Incluindo a autorização

Nós vimos anteriormente o uso do componente *Authentication* para realizarmos a autenticação dos usuários. Também incluímos um evento para garantir que somente usuários autenticados possam acessar a área administrativa. Mas e se precisarmos ter diferentes tipos de usuários administradores? Alguns com permissões diferentes de acesso do que os outros? Esse papel é responsabilidade do componente *Acl*.

ACL (*Access Control List* - lista de controle de acesso) é uma solução simples e flexível para realizar o controle do acesso a determinados recursos.

Alguns conceitos são usados pelo *Acl*:

- papel (*role*): o papel que um usuário desempenha no sistema
- recurso (*resource*): algo a ser protegido
- privilégio (*privilege*): o tipo de acesso exigido

O primeiro passo é o planejamento dos itens citados acima. No nosso projeto, do blog, vamos usar três *roles*:

- visitante: pessoas que não fizeram o login no sistema
- redator: usuários que podem publicar e editar posts, mas não apagá-los
- admin: usuários com todas as permissões de acesso

Vamos ter os seguintes recursos a proteger:

- *Application\Controller\Index*: controlador *IndexController* do módulo *Application*
- *Admin\Controller\Index*: controlador *IndexController* do módulo *Admin*
- *Admin\Controller\Auth*: controlador *AuthController* do módulo *Admin*

Iremos controlar o acesso às *actions* destes controladores.

O primeiro passo é descrever os papéis, recursos e privilégios no nosso arquivo de configuração global da aplicação: *config/autoload/global.php*, incluindo um novo *array*:

```
'acl' => array(
        'roles' => array(
                'visitante'   => null,
                'redator'  => 'visitante',
                'admin' => 'redator'
        ),
        'resources' => array(
                'Application\Controller\Index.index',
                'Admin\Controller\Index.save',
                'Admin\Controller\Index.delete',
                'Admin\Controller\Auth.index',
                'Admin\Controller\Auth.login',
                'Admin\Controller\Auth.logout',
        ),
        'privilege' => array(
                'visitante' => array(
                        'allow' => array(
                                'Application\Controller\Index.index',
                                'Admin\Controller\Auth.index',
                                'Admin\Controller\Auth.login',
                                'Admin\Controller\Auth.logout',
                        )
                ),
                'redator' => array(
                        'allow' => array(
                                'Admin\Controller\Index.save',
                        )
                ),
                'admin' => array(
                        'allow' => array(
                                'Admin\Controller\Index.delete',
                        )
                ),
        )
)
```

https://gist.github.com/4012060

Uma característica interessante do *Acl* é a possibilidade das roles terem herança. Da forma como configuramos, o redator herda as configurações do visitante e o admin herda as configurações do redator. Isso facilita bastante a configuração. Podemos também usar um array *deny* caso seja necessário negar o acesso a determinado recurso. Os recursos estão definidos na forma "Controlador.Acao".

Agora vamos criar uma classe para ler o arquivo de configuração e gerar as *ACLs*. Como as *ACLs* serão usadas em toda a aplicação vamos criar uma classe no módulo *Core*. O conteúdo do arquivo *module\Core\src\Core\Acl\Builder.php* é:

```php
<?php
namespace Core\Acl;

use Zend\ServiceManager\ServiceManager;
use Zend\ServiceManager\ServiceManagerAwareInterface;
use Zend\ServiceManager\Exception\ServiceNotFoundException;

use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;
use Zend\Permissions\Acl\Resource\GenericResource as Resource;

class Builder implements ServiceManagerAwareInterface
{
        /**
        * @var ServiceManager
        */
        protected $serviceManager;

        /**
        * @param ServiceManager $serviceManager
        */
        public function setServiceManager(ServiceManager $serviceManager)
        {
                $this->serviceManager = $serviceManager;
        }

        /**
        * Retrieve serviceManager instance
        *
        * @return ServiceLocatorInterface
        */
        public function getServiceManager()
        {
                return $this->serviceManager;
        }

        /**
        * Constroi a ACL
        * @return Acl
        */
        public function build()
        {
                $config = $this->getServiceManager()->get('Config');
                $acl = new Acl();
                foreach ($config['acl']['roles'] as $role => $parent) {
                        $acl->addRole(new Role($role), $parent);
                }
                foreach ($config['acl']['resources'] as $r) {
                        $acl->addResource(new Resource($r));
```

```
50                  }
51                  foreach ($config['acl']['privilege'] as $role => $privilege) {
52                          if (isset($privilege['allow'])) {
53                                  foreach ($privilege['allow'] as $p) {
54                                          $acl->allow($role, $p);
55                                  }
56                          }
57                          if (isset($privilege['deny'])) {
58                                  foreach ($privilege['deny'] as $p) {
59                                          $acl->deny($role, $p);
60                                  }
61                          }
62                  }
63                  return $acl;
64          }
65  }
```

https://gist.github.com/4012061

A classe *Builder* que criamos acima foi construída para usar as configurações do arquivo *global.php* mas podemos facilmente criar uma nova versão para lermos as configurações de um banco de dados ou arquivo *XML*. Assim podemos facilmente mudar a forma como armazenamos a configuração das *ACLs* sem modificar o resto do sistema.

Vamos agora alterar o evento *mvcPreDispatch* que criamos anteriormente no *module\Admin\Module.php*:

```
1   /**
2    * Verifica se precisa fazer a autorização do acesso
3    * @param  MvcEvent $event Evento
4    * @return boolean
5    */
6   public function mvcPreDispatch($event)
7   {
8       $di = $event->getTarget()->getServiceLocator();
9       $routeMatch = $event->getRouteMatch();
10      $moduleName = $routeMatch->getParam('module');
11      $controllerName = $routeMatch->getParam('controller');
12      $actionName = $routeMatch->getParam('action');
13
14      $authService = $di->get('Admin\Service\Auth');
15      //passa os novos parâmetros
16      if (!
17        $authService->authorize($moduleName, $controllerName, $actionName)
18      ) {
19        throw new \Exception('Você não tem permissão para acessar este recurs\
20  o');
21      }
22
23      return true;
24  }
```

https://gist.github.com/4012065

Agora podemos alterar o serviço, adicionando o método *authorize*:

```
/**
 * Faz a autorização do usuário para acessar o recurso
 * @param string $moduleName Nome do módulo sendo acessado
 * @param string $controllerName Nome do controller
 * @param string $actionName Nome da ação
 * @return boolean
 */
public function authorize($moduleName, $controllerName, $actionName)
{
    $auth = new AuthenticationService();
    $role = 'visitante';

    if ($auth->hasIdentity()) {
        /* pega o role do usuário logado*/
        $session = $this->getServiceManager()->get('Session');
        $user = $session->offsetGet('user');
        $role = $user->role;
    }

    $resource = $controllerName . '.' . $actionName;
     /* monta as acls de acordo com o arquivo de configurações */
    $acl = $this->getServiceManager()
                ->get('Core\Acl\Builder')
                ->build();
        /* verifica se o usuário tem permissão para
            acessar o recurso atual*/
    if ($acl->isAllowed($role, $resource)) {
        return true;
    }
    return false;
}
```

https://gist.github.com/4012068

Vamos alterar também o layout para mostrar a opção de *login/logout* do sistema, para facilitar o acesso aos usuários. Mas para isso vamos precisar criar um *View Helper*, que vai recuperar a *Session* e enviar para a nossa *ViewModel*. Vamos ver isso no próximo tópico.

Enquanto isso, vamos criar alguns usuários de teste na tabela, indicando qual é o papel (redator, visitante ou admin) que cada usuário exerce no sistema. É possível fazer isso usando o *phpMyAdmin*, outra ferramenta gráfica ou com os comandos *SQL* abaixo:

```
1 INSERT INTO users (username, password, name, valid, ROLE)
2           VALUES ('eminetto',md5('teste'),'Elton Minetto', 1, 'admin');
3 INSERT INTO users (username, password, name, valid, ROLE)
4           VALUES ('steve',md5('teste'),'Steve Jobs', 1, 'redator');
5 INSERT INTO users (username, password, name, valid, ROLE)
6           VALUES ('bill',md5('teste'),'Bill Gates', 1, 'visitante');
```

https://gist.github.com/987391

Podemos assim fazer testes e verificar que o acesso aos métodos é controlado de maneira muito rápida e fácil. O uso de *ACLs* permite um controle fácil de ser desenvolvido e com grande versatilidade.

View Helper

Vamos adicionar uma facilidade ao usuário do nosso blog: uma opção para fazer *login* ou *logout* do sistema. Para isso precisamos ter acesso a *Session*, onde armazenamos os dados do usuário, nas nossas visões (incluindo o *layout*). Como a nossa *Session* é um serviço ela não está disponível automaticamente na camada de visão e por isso precisamos de alguma forma fácil de acessar essa funcionalidade. A maneira mais fácil e correta para fazer isso é criarmos um *View Helper*.

Já usamos um *View Helper* nos nossos códigos, como o exemplo:

```
1  <?php
2  echo $this->dateFormat(
3      $post->post_date,
4      \IntlDateFormatter::SHORT,
5      \IntlDateFormatter::SHORT,
6      'pt_BR'
7  );
8  ?>
```

Os *helpers* servem para auxiliar a geração de algum item, principalmente na visão, e existem diversos já existentes no framework, como podemos ver no manual.

Criando um *View Helper*

O primeiro passo é criarmos a classe que vai conter a lógica. Como essa funcionalidade será útil para todos os módulos iremos criá-la dentro do módulo *Core*. Criamos o arquivo *module/Core/src/Core/View/Helper/Session.php*:

```
1  <?php
2
3  namespace Core\View\Helper;
4
5  use Zend\View\Helper\AbstractHelper;
6  use Zend\ServiceManager\ServiceLocatorAwareInterface;
7  use Zend\ServiceManager\ServiceLocatorInterface;
8
9  /**
10  * Helper que inclui a sessão nas views
11  *
12  * @category Application
13  * @package View\Helper
14 * @author  Elton Minetto <eminetto@coderockr.com>
15 */
16 class Session extends AbstractHelper
17         implements ServiceLocatorAwareInterface
18 {
19         /**
20         * Set the service locator.
```

```
21          *
22                  * @param ServiceLocatorInterface $serviceLocator
23          * @return CustomHelper
24          */
25          public function setServiceLocator(ServiceLocatorInterface $serviceLocator)
26          {
27                  $this->serviceLocator = $serviceLocator;
28                  return $this;
29          }
30
31          /**
32          * Get the service locator.
33          *
34          * @return \Zend\ServiceManager\ServiceLocatorInterface
35          */
36          public function getServiceLocator()
37          {
38                  return $this->serviceLocator;
39          }
40
41          public function __invoke()
42          {
43                  $helperPluginManager = $this->getServiceLocator();
44                  $serviceManager = $helperPluginManager->getServiceLocator();
45                  return $serviceManager->get('Session');
46          }
47  }
```

https://gist.github.com/4012071

Alguns comentários:

```
1  class Session extends AbstractHelper
2          implements ServiceLocatorAwareInterface
```

Um *helper* precisa estender a classe *AbstractHelper* para funcionar. Como vamos precisar do *ServiceManager* para acessar a *Session* também precisamos implementar a interface *ServiceLocatorAwareInterface* e dessa forma o framework vai injetar um *ServiceManager* no momento da execução.

```
1  public function __invoke()
```

É nesse método que a lógica do *helper* está contida. Ele será executado quando o invocarmos na visão.

Como o nosso *helper* é customizado, não um dos incluídos no framework, precisamos indicar no nosso arquivo de configurações que ele existe e onde ele encontra-se. Vamos alterar o arquivo *module/Core/config/module.config.php*:

```
<?php
return array(
        'di' => array(),
            'view_helpers' => array(
                'invokables'=> array(
                    'session' => 'Core\View\Helper\Session'
            )
    ),
);
```

https://gist.github.com/4003285

Agora que temos o *helper* habilitado podemos alterar o *layout.phtml* para incluir o uso da sessão:

```
<ul class="nav">
        <li class="active">
                <a href="/"><?php echo $this->translate('Home') ?></a>
        </li>
        <li>
                <?php $user = $this->session()->offsetGet('user'); ?>
                <?php if ($user): ?>
                        <a href="/admin/auth/logout"><?php echo $this->translate('Sair') ?></a>
                <?php else: ?>
                        <a href="/admin/auth/index"><?php echo $this->translate('Entrar') ?></a>
                <?php endif;?>
        </li>
</ul>
```

https://gist.github.com/4003300

Na linha estamos invocando o *helper*, recebendo a instância da sessão que foi salva no nosso serviço de autenticação e usando o método *offsetGet* para recuperar os dados do usuário.

No trecho acima também vemos o uso de outro *helper* o _ translate_. Veremos mais sobre ele em um próximo capítulo.

Cache

Introdução

A técnica de cache é muito usada para melhorar a performance de sites, sejam eles de grande tráfego ou não. Teoricamente quase tudo pode ser armazenado em cache: resultados de consultas, imagens, arquivos css, arquivos js, trechos de código html, etc. A idéia é reaproveitarmos conteúdos já gerados e entregá-los rapidamente ao usuário. Um exemplo clássico é armazenar o resultado de uma consulta ao banco de dados, para não precisar acessar o banco todas as vezes.

No Zend Framework o cache é fornecido usando-se as classes do namespace *Zend\Cache*. O cacheamento é fornecido ao programador através de *"adapters"*.

Os principais adapters disponíveis são:

- *Apc*: usa o cache em memória fornecido pela extensão *APC (Alternative PHP Cache)* do PHP
- *Filesystem*: os dados do cache são armazenados em arquivos no sistema operacional
- *Memcached*: os dados serão salvos no *Memcached*, um servidor específico para cache, usado por grandes arquiteturas como o *Facebook*
- *Memory*: salva o cache na memória, na forma de *arrays*.

Também é possível criarmos nossos próprios *adapters*, apesar de ser necessário em poucos casos.

Vamos ver alguns exemplos de uso. O primeiro passo é criarmos entradas no nosso arquivo de configurações, no *application.config.php*:

```
1 'cache' => array(
2     'adapter' => 'memory'
3 ),
```

Cada *adapter* possui configurações especiais que podem mudar seu comportamento padrão. Mais detalhes podem ser encontrados no manual oficial do framework.

Configurando o Cache

A forma mais fácil de acessar o cache é usando o *ServiceManager*, para que as dependências de criação sejam facilmente criadas sempre que precisarmos. Para isso vamos configurar o nosso módulo *Admin* para que seja registrado o serviço Cache. No arquivo *module/Admin/config/module.config.php* vamos adicionar:

```
'service_manager' => array(
    'factories' => array(
        'Session' => function($sm) {
            return new Zend\Session\Container('ZF2napratica');
        },
        'Admin\Service\Auth' => function($sm) {
            $dbAdapter = $sm->get('DbAdapter');
            return new Admin\Service\Auth($dbAdapter);
        },
        'Cache' => function($sm) {
            $config = include __DIR__ .
                                                '/../../../config/application.config.php';
            $cache = StorageFactory::factory(
                array(
                    'adapter' => $config['cache']['adapter'],
                    'plugins' => array(
                        'exception_handler' =>
                            array('throw_exceptions' => false),
                        'Serializer'
                    ),
                )
            );

            return $cache;
        }
    )
),
```

https://gist.github.com/4012076

Usando o cache

Com o serviço de cache registrado no *ServiceManager* podemos usá-lo em controladores ou serviços, conforme o exemplo abaixo.

```
$cache = $this->getServiceManager()->get('Cache');
ou
$cache = $this->getServiceLocator()->get('Cache');
```

Para adicionar um item ao cache basta:

```
$cache->addItem('chave', $valor);
```

Desta forma estamos incluindo no cache o conteúdo da variável *$valor* com o nome *'chave'*. O conteúdo da variável *$valor* vai ser serializado automaticamente, sendo ele uma string, inteiro ou mesmo objeto. A única ressalva é que alguns *adapters* tem restrições quanto ao que podem armazenar, sendo indicado uma leitura no manual caso algum item não consiga ser salvo no cache.

Para recuperarmos esse valor usamos:

```
$cache->getItem('chave');
```

Vários componentes do framework possuem integração com o cache, bastando indicar que estamos usando um cache. Um exemplo disso é o *Paginator* que estamos usando em nosso *Application\Controller\IndexController*. Podemos alterá-lo para usar o cache, conforme o exemplo:

```
$paginator = new Paginator($paginatorAdapter);
$cache = $this->getServiceLocator()->get('Cache');
$paginator->setCache($cache);
```

OBS: Não são todos os *adapters* que podem ser usados nesse exemplo, com o *Paginator*. No momento da escrita desse texto apenas o *Filesystem* e o *Memory* são suportados.

Traduções

Traduzindo o projeto

É cada vez mais comum a criação de projetos de software que precisem atender a públicos de países diferentes, com línguas distintas. O Zend Framework possui um componente de tradução que facilita a criação deste tipo de projeto, o *Zend\I18n\Translator\Translator*. Para fazer uso do componente vamos inicialmente criar um diretório no projeto onde iremos armazenar os arquivos das traduções, o *module/Application/language*.

As traduções podem ser salvas usando-se um dos padrões suportados:

- *PHP arrays*
- *Gettext*
- *Tmx*
- *Xliff* Vamos adicionar as linhas abaixo na configuração do módulo, no arquivo *module/Application/config/module.config.php*:

```
1  'service_manager' => array(
2       'factories' => array(
3              'translator' => 'Zend\I18n\Translator\TranslatorServiceFactory'\
4  ,
5          ),
6      ),
7      'translator' => array(
8          'locale' => 'pt_BR',
9          'translation_file_patterns' => array(
10             array(
11             'type'     => 'phparray',
12             'base_dir' => __DIR__ . '/../language',
13             'pattern'  => '%s.php',
14         ),
15     ),
16 ),
```

https://gist.github.com/4012079

Vamos agora criar o arquivo de traduções para o português do Brasil: *language/pt_BR.php*:

```
1  <?php
2  return array(
3      'Home' => 'Página inicial',
4      'logout' => 'Sair'
5  );
```

Podemos ter arquivos para cada língua, e mudar a configuração do *Translator* conforme a seleção do usuário ou a URL acessada.

Para usar as traduções vamos usar o *View Helper translate()* na nossa view ou layout. No arquivo *layout.phtml* podemos ter o seguinte código:

```
<li class="active">
    <a href="/"><?php echo $this->translate('Home') ?></a>
</li>
```

Caso a chave *Home* exista no arquivo de traduções ela será substituída pelo conteúdo traduzido e caso não exista a própria palavra *Home* vai ser exibida, sem causar erros para o usuário.

Traduzindo formulários

Também podemos usar o *translator* para traduzir as mensagens de validação de nossos formulários. No *Admin\Controller\IndexController* usamos a validação da entidade *Post* para validar nosso formulário:

```
$form->setInputFilter($post->getInputFilter());
```

Podemos indicar que os filtros devem usar um *translator* para traduzir as mensagens de erro. No controlador adicionamos, antes da criação do formulário:

```
$translator = $this->getServiceLocator()
                   ->get('translator');
\Zend\Validator\AbstractValidator::setDefaultTranslator($translator);
```

Precisamos também adicionar as mensagens em português no nosso arquivo *pt_BR.php*:

```
<?php
return array(
        'Home' => 'Página inicial',
        'logout' => 'Sair',
        'notEmptyInvalid' => 'Valor nulo ou inválido',
        'isEmpty' => 'Valor obrigatório e não pode ser nulo',
);
```

As chaves *notEmptyInvalid* e *isEmpty* encontram-se documentadas no manual, ou diretamente no código do validador, no caso o arquivo *vendor/zendframework/zendframework/library/Zend/Validator/-NotEmpty.php*

Outra funcionalidade interessante do *Translator* é a possibilidade de usar cache. Assim as traduções são armazenadas no cache, não necessitando que o arquivo seja aberto e processado todas as vezes que for usado. Para isso basta usar o método *setCache()* conforme o exemplo:

```
$translator = $this->getServiceLocator()->get('translator');
$cache = $this->getServiceLocator()->get('Cache');
$translator->setCache($cache);
```

Requisições Assíncronas

Uma das funcionalidades mais usadas em projetos web nos últimos anos é a possibilidade dos navegadores, com auxílio do JavaScript, executarem várias requisições assíncronas e simultâneas, o famoso *AJAX*.

Usar essa característica no Zend Framework é muito simples, basta modificar a forma como as *actions* do controlador acessado renderizam seu resultado.

Gerando uma API de comentários

Vamos criar um exemplo.

No *Application\Controller\IndexController* vamos criar uma *action* chamada *commentsAction* que irá receber o código de um *Post* e retornar todos os seus comentários, em um formato *JSON* para ser usado em um código *JavaScript*, por exemplo.

```
1  /**
2   * Retorna os comentários de um post
3   * @return Zend\Http\Response
4   */
5  public function commentsAction()
6  {
7      $id = (int) $this->params()->fromRoute('id', 0);
8      $where = array('post_id' => $id);
9      $comments = $this->getTable('Application\Model\Comment')
10                      ->fetchAll(null, $where)
11                      ->toArray();
12
13     $response = $this->getResponse();
14     $response->setStatusCode(200);
15     $response->setContent(json_encode($comments));
16     $response->getHeaders()
17              ->addHeaderLine('Content-Type', 'application/json');
18     return $response;
19 }
```

https://gist.github.com/4012081

O que esta *action* faz agora é retornar um objeto *Response* e não mais uma *ViewModel*, como é seu comportamento padrão. Desta forma nenhuma tela é renderizada e podemos indicar em detalhes o formato do retorno. Desta forma fica muito fácil usar em scripts ou outros componentes.

OBS: Antes de testarmos o acesso precisamos adicionar este recurso na nossa configuração de *ACLs*, conforme configuramos no capítulo correspondente. É necessário incluir a linhas abaixo, no arquivo *global.php*, no array *resources* e no array *privilege['visitante']['allow']*:

```
1  'Application\Controller\Index.comments'
```

Mostrando a view sem o layout

Outro caso comum é quando nossa requisição precisa retornar o *HTML* da view, mas não usar o *layout*. Nestes casos mudaríamos o código para:

```
/**
 * Retorna os comentários de um post
 * @return Zend\Http\Response
 */
public function commentsAction()
{
        $id = (int) $this->params()->fromRoute('id', 0);
        $where = array('post_id' => $id);
        $comments = $this->getTable('Application\Model\Comment')
                    ->fetchAll(null, $where)
                    ->toArray();
        $result = new ViewModel(array(
                'comments' => $comments
                )
        );
        $result->setTerminal(true);
        return $result;
}
```

https://gist.github.com/4012084

Precisamos também criar uma *view* para que o conteúdo seja renderizado. O conteúdo do *view/application/index/comments.phtml* poderia ser:

```
<?php var_dump($comments);?>
```

Neste exemplo iremos gerar o conteúdo dos comentários, mas não executamos o *layout* padrão, tendo como resposta um *HTML* puro.

Com estes dois exemplos podemos ver a facilidade para criar *actions* que podem ser acessadas usando requisições assíncronas.

Doctrine

O Doctrine é um dos projetos mais reconhecidos e respeitados na comunidade de desenvolvedores PHP. Trata-se de um *ORM* (*Object-Relational Mapping*), ferramenta que facilita o uso de bases de dados relacionais em um ambiente orientado a objetos, como o usado no nosso projeto. Mais teorias sobre *ORM* podem ser encontradas na internet e no próprio site do Doctrine.

Neste capítulo não vamos nos aprofundar sobre o Doctrine, pois isso é assunto para um novo livro. O que vamos ver aqui é como integrarmos esse novo componente ao nosso ambiente.

O *Doctrine* pode ser usado como substituto do nosso *TableGateway* e das classes do *namespace Zend\Db*. O fato de usarmos o *TableGateway* no projeto vai nos facilitar bastante a integração, pois os conceitos são bem parecidos.

Instalando o Doctrine

Vamos usar o *Composer* para instalar o Doctrine e suas dependências. Para isso precisamos alterar o *composer.json*:

```
{
        "name": "zendframework/skeleton-application",
        "description": "Skeleton Application for ZF2",
        "license": "BSD-3-Clause",
        "keywords": [
                "framework",
                "zf2"
        ],
        "homepage": "http://framework.zend.com/",
        "minimum-stability": "dev",
        "require": {
                "php": ">=5.3.3",
                "zendframework/zendframework": "2.*",
                "doctrine/doctrine-orm-module": "dev-master"
        }
}
```

https://gist.github.com/4038409

Foram adicionadas as linhas

```
"minimum-stability": "dev",
```

e

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

Agora basta executar o comando *php composer.phar install* para que tudo seja instalado.

Configurando o projeto

Vamos configurar no *config/global.php* os detalhes da conexão com o banco de dados:

```
'doctrine' => array(
        'connection' => array(
                'driver'   => 'pdo_mysql',
                'host'     => 'localhost',
                'port'     => '3306',
                'user'     => 'zend',
                'password' => 'zend',
                'dbname'   => 'zf2napratica'
        )
),
```

https://gist.github.com/4038417

Podemos configurar o *Doctrine* em um dos módulos ou em um dos arquivos de configuração da aplicação (*application.config.php* ou *global.php*). Vamos alterar nesse exemplo o *module/Admin/config/-module.config.php*. O novo arquivo ficou:

```
<?php
namespace Admin;

// module/Admin/config/module.config.php:
return array(
    'controllers' => array( //add module controllers
        'invokables' => array(
            'Admin\Controller\Index' => 'Admin\Controller\IndexController',
            'Admin\Controller\Auth' => 'Admin\Controller\AuthController',
            'Admin\Controller\User' => 'Admin\Controller\UserController',
        ),
    ),

    'router' => array(
        'routes' => array(
            'admin' => array(
                'type'    => 'Literal',
                'options' => array(
                    'route'    => '/admin',
                    'defaults' => array(
                        '__NAMESPACE__' => 'Admin\Controller',
                        'controller'    => 'Index',
                        'action'        => 'index',
                        'module'        => 'admin'
                    ),
                ),
                'may_terminate' => true,
                'child_routes' => array(
```

```
29                      'default' => array(
30                          'type'    => 'Segment',
31                          'options' => array(
32                              'route'    => '/[:controller[/:action]]',
33                              'constraints' => array(
34                                  'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
35                                  'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
36                              ),
37                              'defaults' => array(
38                              ),
39                          ),
40                          //permite mandar dados pela url
41                          'child_routes' => array(
42                              'wildcard' => array(
43                                  'type' => 'Wildcard'
44                              ),
45                          ),
46                      ),
47
48                  ),
49              ),
50          ),
51      ),
52      'view_manager' => array( //the module can have a specific layout
53          // 'template_map' => array(
54          //     'layout/layout' => __DIR__ . '/../view/layout/layout.phtml',
55          // ),
56          'template_path_stack' => array(
57              'admin' => __DIR__ . '/../view',
58          ),
59      ),
60      // 'db' => array( //module can have a specific db configuration
61      //     'driver' => 'PDO_SQLite',
62      //     'dsn' => 'sqlite:' . __DIR__ .'/../data/skel.db',
63      //     'driver_options' => array(
64      //         PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
65      //     )
66      // )
67      'service_manager' => array(
68          'factories' => array(
69              'Session' => function($sm) {
70                  return new \Zend\Session\Container('ZF2napratica');
71              },
72              'Admin\Service\Auth' => function($sm) {
73                  $dbAdapter = $sm->get('DbAdapter');
74                  return new Service\Auth($dbAdapter);
75              },
76              'Cache' => function($sm) {
77                  $config = $sm->get('Configuration');
```

```
78                  $cache = StorageFactory::factory(
79                      array(
80                          'adapter' => $config['cache']['adapter'],
81                          'plugins' => array(
82                              'exception_handler' => array('throw_exceptions'\
83   => false),
84                              'Serializer'
85                          ),
86                      )
87                  );
88
89                  return $cache;
90              },
91              'Doctrine\ORM\EntityManager' => function($sm) {
92                  $config = $sm->get('Configuration');
93
94                  $doctrineConfig = new \Doctrine\ORM\Configuration();
95                  $cache = new $config['doctrine']['driver']['cache'];
96                  $doctrineConfig->setQueryCacheImpl($cache);
97                  $doctrineConfig->setProxyDir('/tmp');
98                  $doctrineConfig->setProxyNamespace('EntityProxy');
99                  $doctrineConfig->setAutoGenerateProxyClasses(true);
100
101                 $driver = new \Doctrine\ORM\Mapping\Driver\AnnotationDriver\
102 (
103                     new \Doctrine\Common\Annotations\AnnotationReader(),
104                     array($config['doctrine']['driver']['paths'])
105                 );
106                 $doctrineConfig->setMetadataDriverImpl($driver);
107                 $doctrineConfig->setMetadataCacheImpl($cache);
108                 \Doctrine\Common\Annotations\AnnotationRegistry::registerFi\
109 le(
110                     getenv('PROJECT_ROOT') .
111                     '/vendor/doctrine/orm/lib/Doctrine/ORM/Mapping/Driver/D\
112 octrineAnnotations.php'
113                 );
114                 $em = \Doctrine\ORM\EntityManager::create(
115                     $config['doctrine']['connection'],
116                     $doctrineConfig
117                 );
118                 return $em;
119
120             },
121         )
122     ),
123     'doctrine' => array(
124         'driver' => array(
125             'cache' => 'Doctrine\Common\Cache\ArrayCache',
126             'paths' => array(__DIR__ . '/../src/' . __NAMESPACE__ . '/Model\
```

```
')
            ),
    )
);
```

https://gist.github.com/4038412

Primeiro precisamos colocar o *namespace* no arquivo de configuração (vamos precisar alterar isso nos arquivos *module.config.php* de todos os módulos que irão usar o *Doctrine*):

```
namespace Admin;
```

Também adicionamos o *array _doctrine* com as configurações de cache e o caminho das nossas entidades.

No *array service_manager* incluimos um novo serviço, chamado *Doctrine\ORM\EntityManager* que iremos usar sempre que precisarmos manipular alguma das nossas entidades. Desta forma toda a configuração do *Doctrine* é feita pelo *ServiceManager*.

O *EntityManager* é o componente do Doctrine que substitui o *TableGateway* que usamos até agora no projeto.

Criando uma entidade

Para usarmos o Doctrine precisamos alterar as nossas entidades. Vamos usar como exemplo a entidade *User* que foi reescrita da seguinte forma:

```
<?php
namespace Admin\Model;

use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;
use Core\Model\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* Entidade User
*
* @category Admin
* @package Model
*
* @ORM\Entity
* @ORM\Table(name="users")
*/
class User extends Entity
{

```

```
24          /**
25          * @ORM\Id
26          * @ORM\Column(type="integer");
27          * @ORM\GeneratedValue(strategy="AUTO")
28          */
29          protected $id;
30
31          /**
32          * @ORM\Column(type="string")
33          */
34          protected $username;
35
36          /**
37          * @ORM\Column(type="string")
38          */
39          protected $password;
40
41          /**
42          * @ORM\Column(type="string")
43          */
44          protected $name;
45
46          /**
47          * @ORM\Column(type="integer");
48          */
49          protected $valid;
50
51          /**
52          * @ORM\Column(type="string")
53          */
54          protected $role;
55
56          /**
57          * Configura os filtros dos campos da entidade
58          *
59          * @return Zend\InputFilter\InputFilter
60          */
61          public function getInputFilter()
62          {
63                  /* Continua o mesmo codigo apresentado no capítulo 'Modulos'*/
64          }
65  }
```

https://gist.github.com/4038434

A primeira mudança é a inclusão de um componente do Doctrine:

```
use Doctrine\ORM\Mapping as ORM;
```

Neste exemplo estamos usando o recurso de *Annotations* para descrever a entidade e sua ligação com o banco de dados, como o *@ORM\Entity* e *@ORM\Table(name="users")*. O *Doctrine* permite que façamos essa descrição de outras formas (*XML* ou *YML*), além de possuir mais opções (como a descrição das chaves estrangeiras) que estão disponíveis na documentação oficial;

Essas são as únicas alterações que precisamos fazer para a entidade ser compatível com o *Doctrine*. Iremos usar a mesma forma de validar os campos apresentada nos capítulos anteriores.

Criando os testes

A primeira tarefa que precisamos fazer é alterar o *config/test.config.php* para incluir a configuração do Doctrine. Vamos deixar a configuração do Zend (*array db*) para não quebrarmos os testes anteriores.

```
<?php
return array(
      'db' => array(
        'driver' => 'PDO',
        'dsn'    => 'mysql:dbname=zf2napratica_test;host=localhost',
        'username' => 'zend',
        'password' => 'zend',
        'driver_options' => array(
          PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
    'doctrine' => array(
        'connection' => array(
          'driver'   => 'pdo_mysql',
          'host'     => 'localhost',
          'port'     => '3306',
          'user'     => 'zend',
          'password' => 'zend',
          'dbname'   => 'zf2napratica_test'
        )
      ),
  );
```

https://gist.github.com/4051966

Agora podemos alterar o *module/Admin/tests/src/Admin/Model/UserTest.php* para usar o *EntityManager*:

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

use Core\Test\ModelTestCase;
use Admin\Model\User;
use Zend\InputFilter\InputFilterInterface;

/**
 * @group Model
 */
class UserTest extends ModelTestCase
{
    /**
     * EntityManager
     * @var Doctrine\ORM\EntityManager
     */
    private $em;

    public function setup()
    {
        parent::setup();
        $this->em = $this->serviceManager
                         ->get('Doctrine\ORM\EntityManager');
    }

    public function testGetInputFilter()
    {
        $user = new User();
        $if = $user->getInputFilter();
        //testa se existem filtros
        $this->assertInstanceOf("Zend\InputFilter\InputFilter", $if);
        return $if;
    }

    /**
     * @depends testGetInputFilter
     */
    public function testInputFilterValid($if)
    {
        //testa os filtros
        $this->assertEquals(6, $if->count());

        $this->assertTrue($if->has('id'));
        $this->assertTrue($if->has('username'));
        $this->assertTrue($if->has('password'));
        $this->assertTrue($if->has('name'));
        $this->assertTrue($if->has('valid'));
        $this->assertTrue($if->has('role'));
    }
```

```
50
51      /**
52       * @expectedException Core\Model\EntityException
53       */
54      public function testInputFilterInvalidoUsername()
55      {
56          //testa se os filtros estão funcionando
57          $user = new User();
58          //username só pode ter 50 caracteres
59          $user->username = 'Lorem Ipsum e simplesmente uma simulacao de text\
60  o da industria tipografica e de impressos. Lorem Ipsum e simplesmente uma s\
61  imulacao de texto da industria tipografica e de impressos';
62      }
63
64      /**
65       * @expectedException Core\Model\EntityException
66       */
67      public function testInputFilterInvalidoRole()
68      {
69          //testa se os filtros estão funcionando
70          $user = new User();
71          //role só pode ter 20 caracteres
72          $user->role = 'Lorem Ipsum é simplesmente uma simulação de texto da\
73   indústria
74          tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulaçã\
75  o de texto
76          da indústria tipográfica e de impressos';
77      }
78
79      /**
80       * Teste de inserção de um user válido
81       */
82      public function testInsert()
83      {
84          $user = $this->addUser();
85
86          //testa o filtro de tags e espaços
87          $this->assertEquals('Steve Jobs', $user->name);
88          //testa o auto increment da chave primária
89          $this->assertEquals(1, $user->id);
90      }
91
92      /**
93       * @expectedException Core\Model\EntityException
94       * @expectedExceptionMessage Input inválido: username =
95       */
96      public function testInsertInvalido()
97      {
98          $user = new user();
```

```
99          $user->name = 'teste';
100         $user->username = '';
101
102         $this->em->persist($user);
103         $this->em->flush();
104     }
105
106     public function testUpdate()
107     {
108         $user = $this->addUser();
109
110         $id = $user->id;
111
112         $this->assertEquals(1, $id);
113
114         $user = $this->em->find('Admin\Model\User', $id);
115         $this->assertEquals('Steve Jobs', $user->name);
116
117         $user->name = 'Bill <br>Gates';
118         $this->em->persist($user);
119         $this->em->flush();
120
121         $user = $this->em->find('Admin\Model\User', $id);
122         $this->assertEquals('Bill Gates', $user->name);
123     }
124
125     public function testDelete()
126     {
127         $user = $this->addUser();
128
129         $id = $user->id;
130
131         $user = $this->em->find('Admin\Model\User', $id);
132         $this->em->remove($user);
133         $this->em->flush();
134
135         $user = $this->em->find('Admin\Model\User', $id);
136         $this->assertNull($user);
137     }
138
139     private function addUser()
140     {
141         $user = new User();
142         $user->username = 'steve';
143         $user->password = md5('apple');
144         $user->name = 'Steve <b>Jobs</b>';
145         $user->valid = 1;
146         $user->role = 'admin';
147
```

```
148         $this->em->persist($user);
149         $this->em->flush();
150 
151         return $user;
152     }
153 
154 }
```

https://gist.github.com/4051972

As alterações no teste foram poucas, basicamente substituir o *TableGateway* pelo *EntityManager* e mudar alguns *assertions*. A estrutura continou a mesma, incluindo os testes dos filtros e validações.

CRUD de Usuário

Vamos agora aplicar o *Doctrine* no controlador, para fazermos a gerência dos usuários.

Teste do Controlador

Primeiro, o teste, no *module/Admin/tests/src/Admin/Controller/UserControllerTest.php*:

```
1  <?php
2  use Core\Test\ControllerTestCase;
3  use Admin\Controller\IndexController;
4  use Admin\Model\User;
5  use Zend\Http\Request;
6  use Zend\Stdlib\Parameters;
7  use Zend\View\Renderer\PhpRenderer;
8  
9  
10 /**
11  * @group Controller
12  */
13 class UserControllerTest extends ControllerTestCase
14 {
15     /**
16      * Namespace completa do Controller
17      * @var string
18      */
19     protected $controllerFQDN = 'Admin\Controller\UserController';
20 
21     /**
22      * Nome da rota. Geralmente o nome do módulo
23      * @var string
24      */
25     protected $controllerRoute = 'admin';
26 
27     /**
```

```
28      * Testa a página inicial, que deve mostrar os posts
29      */
30     public function testUserIndexAction()
31     {
32         // Cria users para testar
33         $userA = $this->addUser();
34         $userB = $this->addUser();
35 
36         // Invoca a rota index
37         $this->routeMatch->setParam('action', 'index');
38         $result = $this->controller->dispatch(
39           $this->request, $this->response
40         );
41 
42         // Verifica o response
43         $response = $this->controller->getResponse();
44         $this->assertEquals(200, $response->getStatusCode());
45 
46         // Testa se um ViewModel foi retornado
47         $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
48 
49         // Testa os dados da view
50         $variables = $result->getVariables();
51 
52         $this->assertArrayHasKey('users', $variables);
53 
54         // Faz a comparação dos dados
55         $controllerData = $variables["users"];
56         $this->assertEquals($userA->name, $controllerData[0]->name);
57         $this->assertEquals($userB->name, $controllerData[1]->name);
58     }
59 
60     /**
61      * Testa a tela de inclusão de um novo registro
62      * @return void
63      */
64     public function testUserSaveActionNewRequest()
65     {
66 
67         // Dispara a ação
68         $this->routeMatch->setParam('action', 'save');
69         $result = $this->controller->dispatch(
70           $this->request, $this->response
71         );
72         // Verifica a resposta
73         $response = $this->controller->getResponse();
74         $this->assertEquals(200, $response->getStatusCode());
75 
76         // Testa se recebeu um ViewModel
```

```
77          $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
78
79          //verifica se existe um form
80          $variables = $result->getVariables();
81          $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
82          $form = $variables['form'];
83          //testa os ítens do formulário
84          $id = $form->get('id');
85          $this->assertEquals('id', $id->getName());
86          $this->assertEquals('hidden', $id->getAttribute('type'));
87      }
88
89      /**
90       * Testa a tela de alteração de um user
91       */
92      public function testUserSaveActionUpdateFormRequest()
93      {
94          $userA = $this->addUser();
95
96          // Dispara a ação
97          $this->routeMatch->setParam('action', 'save');
98          $this->routeMatch->setParam('id', $userA->id);
99          $result = $this->controller->dispatch(
100           $this->request, $this->response
101         );
102
103         // Verifica a resposta
104         $response = $this->controller->getResponse();
105         $this->assertEquals(200, $response->getStatusCode());
106
107         // Testa se recebeu um ViewModel
108         $this->assertInstanceOf('Zend\View\Model\ViewModel', $result);
109
110         $variables = $result->getVariables();
111
112         //verifica se existe um form
113         $variables = $result->getVariables();
114         $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
115         $form = $variables['form'];
116
117         //testa os ítens do formulário
118         $id = $form->get('id');
119         $name = $form->get('name');
120         $this->assertEquals('id', $id->getName());
121         $this->assertEquals($userA->id, $id->getValue());
122         $this->assertEquals($userA->name, $name->getValue());
123     }
124
125     /**
```

```
126      * Testa a inclusão de um novo user
127      */
128     public function testUserSaveActionPostRequest()
129     {
130         // Dispara a ação
131         $this->routeMatch->setParam('action', 'save');
132 
133         $this->request->setMethod('post');
134         $this->request->getPost()->set('name', 'Bill Gates');
135         $this->request->getPost()->set('password', md5('apple'));
136         $this->request->getPost()->set('username', 'bill');
137         $this->request->getPost()->set('valid', 1);
138         $this->request->getPost()->set('role', 'admin');
139 
140         $result = $this->controller->dispatch(
141           $this->request, $this->response
142         );
143         // Verifica a resposta
144         $response = $this->controller->getResponse();
145         //a página redireciona, então o status = 302
146         $this->assertEquals(302, $response->getStatusCode());
147         $headers = $response->getHeaders();
148         $this->assertEquals('Location: /admin/user', $headers->get('Locatio\
149 n'));
150 
151     }
152 
153     public function testUserUpdateAction()
154     {
155         $user = $this->addUser();
156         // Dispara a ação
157         $this->routeMatch->setParam('action', 'save');
158 
159         $this->request->setMethod('post');
160         $this->request->getPost()->set('id', $user->id);
161         $this->request->getPost()->set('name', 'Alan Turing');
162         $this->request->getPost()->set('password', md5('apple'));
163         $this->request->getPost()->set('username', 'bill');
164         $this->request->getPost()->set('valid', 1);
165         $this->request->getPost()->set('role', 'admin');
166 
167         $result = $this->controller->dispatch(
168           $this->request, $this->response
169         );
170 
171         $response = $this->controller->getResponse();
172         //a página redireciona, então o status = 302
173         $this->assertEquals(302, $response->getStatusCode());
174         $headers = $response->getHeaders();
```

```
175         $this->assertEquals(
176           'Location: /admin/user', $headers->get('Location')
177         );
178     }
179 
180     /**
181      * Tenta salvar com dados inválidos
182      *
183      */
184     public function testUserSaveActionInvalidPostRequest()
185     {
186         // Dispara a ação
187         $this->routeMatch->setParam('action', 'save');
188 
189         $this->request->setMethod('post');
190         $this->request->getPost()->set('username', '');
191 
192         $result = $this->controller->dispatch(
193           $this->request, $this->response
194         );
195 
196         //verifica se existe um form
197         $variables = $result->getVariables();
198         $this->assertInstanceOf('Zend\Form\Form', $variables['form']);
199         $form = $variables['form'];
200 
201         //testa os errors do formulário
202         $username = $form->get('username');
203         $usernameErrors = $username->getMessages();
204         $this->assertEquals(
205           "Value is required and can't be empty", $usernameErrors['isEmpty'\
206 ]
207         );
208 
209     }
210 
211     /**
212      * Testa a exclusão sem passar o id do user
213      * @expectedException Exception
214      * @expectedExceptionMessage Código obrigatório
215      */
216     public function testUserInvalidDeleteAction()
217     {
218         // Dispara a ação
219         $this->routeMatch->setParam('action', 'delete');
220 
221         $result = $this->controller->dispatch(
222           $this->request, $this->response
223         );
```

```
224         // Verifica a resposta
225         $response = $this->controller->getResponse();
226 
227     }
228 
229     /**
230      * Testa a exclusão do user
231      */
232     public function testUserDeleteAction()
233     {
234         $user = $this->addUser();
235         // Dispara a ação
236         $this->routeMatch->setParam('action', 'delete');
237         $this->routeMatch->setParam('id', $user->id);
238 
239         $result = $this->controller->dispatch(
240           $this->request, $this->response
241         );
242         // Verifica a resposta
243         $response = $this->controller->getResponse();
244         //a página redireciona, então o status = 302
245         $this->assertEquals(302, $response->getStatusCode());
246         $headers = $response->getHeaders();
247         $this->assertEquals(
248           'Location: /admin/user', $headers->get('Location')
249         );
250     }
251 
252     /**
253      * Adiciona um user para os testes
254      */
255     private function addUser()
256     {
257         $user = new User();
258         $user->username = 'steve';
259         $user->password = md5('apple');
260         $user->name = 'Steve <b>Jobs</b>';
261         $user->valid = 1;
262         $user->role = 'admin';
263 
264         $em = $this->serviceManager->get('Doctrine\ORM\EntityManager');
265         $em->persist($user);
266         $em->flush();
267 
268         return $user;
269     }
270 
271 }
```

https://gist.github.com/4052198

O teste também não teve muitas alterações em comparação com os demais testes de controladores.

Formulário de inclusão de usuário

Conforme vimos no teste criado precisamos um formulário para efetuar a inclusão/alteração do novo usuário. O arquivo *module/Admin/src/Admin/Form/User.php* ficou desta forma:

```
<?php
namespace Admin\Form;

use Zend\Form\Form;

class User extends Form
{
    public function __construct()
    {
        parent::__construct('user');
        $this->setAttribute('method', 'post');
        $this->setAttribute('action', '/admin/user/save');

        $this->add(array(
            'name' => 'id',
            'attributes' => array(
                'type'  => 'hidden',
            ),
        ));

        $this->add(array(
            'name' => 'name',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Nome',
            ),
        ));

        $this->add(array(
            'name' => 'username',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Username',
            ),
        ));

```

```
41          $this->add(array(
42              'name' => 'password',
43              'attributes' => array(
44                  'type'  => 'password',
45              ),
46              'options' => array(
47                  'label' => 'Senha',
48              ),
49          ));
50
51          $this->add(array(
52              'name' => 'role',
53              'attributes' => array(
54                  'type'  => 'text',
55              ),
56              'options' => array(
57                  'label' => 'Papel',
58              ),
59          ));
60
61          $this->add(array(
62              'name' => 'submit',
63              'attributes' => array(
64                  'type'  => 'submit',
65                  'value' => 'Enviar',
66                  'id' => 'submitbutton',
67              ),
68          ));
69      }
70  }
```

https://gist.github.com/4038450

Criando o controlador

Vamos agora criar o controlador para gerenciarmos a entidade *User* com o *EntityManager*. O arquivo *module/Admin/src/Admin/Controller/UserController*:

```
1   <?php
2   namespace Admin\Controller;
3
4   use Zend\View\Model\ViewModel;
5   use Core\Controller\ActionController;
6   use Admin\Model\User;
7   use Admin\Form\User as UserForm;
8
9   use Doctrine\ORM\EntityManager;
10
11  /**
```

```
12  * Controlador que gerencia os posts
13  *
14  * @category Admin
15  * @package Controller
16  * @author  Elton Minetto <eminetto@coderockr.com>
17  */
18  class UserController extends ActionController
19  {
20
21      /**
22      * @var Doctrine\ORM\EntityManager
23      */
24      protected $em;
25
26      public function setEntityManager(EntityManager $em)
27      {
28          $this->em = $em;
29      }
30
31      public function getEntityManager()
32      {
33          if (null === $this->em) {
34              $this->em = $this->getServiceLocator()
35                               ->get('Doctrine\ORM\EntityManager');
36          }
37          return $this->em;
38      }
39
40      /**
41      * Mostra os usuário cadastrados
42      * @return void
43      */
44      public function indexAction()
45      {
46          $users = $this->getEntityManager()
47                        ->getRepository('Admin\Model\User')
48                        ->findAll();
49          return new ViewModel(array(
50              'users' => $users
51          ));
52      }
53
54      /**
55      * Cria ou edita um user
56      * @return void
57      */
58      public function saveAction()
59      {
60          $form = new UserForm();
```

```
61          $request = $this->getRequest();
62          if ($request->isPost()) {
63              $user = new User;
64              $form->setInputFilter($user->getInputFilter());
65              $form->setData($request->getPost());
66              if ($form->isValid()) {
67                  $data = $form->getData();
68                  unset($data['submit']);
69                  $data['valid'] = 1;
70                  $data['password'] = md5($data['password']);
71                  $user->setData($data);
72
73                  $this->getEntityManager()->persist($user);
74                  $this->getEntityManager()->flush();
75
76                  return $this->redirect()->toUrl('/admin/user');
77              }
78          }
79          $id = (int) $this->params()->fromRoute('id', 0);
80          if ($id > 0) {
81              $user = $this->getEntityManager()
82                           ->find('Admin\Model\User', $id);
83              $form->bind($user);
84              $form->get('submit')->setAttribute('value', 'Edit');
85          }
86          return new ViewModel(
87              array('form' => $form)
88          );
89      }
90
91      /**
92      * Exclui um post
93      * @return void
94      */
95      public function deleteAction()
96      {
97          $id = (int) $this->params()->fromRoute('id', 0);
98          if ($id == 0) {
99              throw new \Exception("Código obrigatório");
100         }
101
102         $user = $this->getEntityManager()->find('Admin\Model\User', $id);
103         if ($user) {
104             $this->getEntityManager()->remove($user);
105             $this->getEntityManager()->flush();
106         }
107         return $this->redirect()->toUrl('/admin/user');
108     }
109 }
```

https://gist.github.com/4038454

Novamente tivemos poucas alterações para incluirmos o *EntityManager* no processo. A principal mudança é na forma como o *Doctrine* faz a alteração do registro, necessitando primeiro recuperá-lo do banco de dados (*$user = $this->getEntityManager()->find('Admin\Model\User', $id);*) para depois alterá-lo com o comando *persist*. O mesmo vale para o processo de remoção de um registro.

OBS: Antes de testarmos o acesso precisamos adicionar este recurso na nossa configuração de *ACLs*, conforme configuramos no capítulo correspondente. É necessário incluir a linhas abaixo, no arquivo *global.php*, no array *resources* e no array *privilege['visitante']['allow']* (ou outra *role* escolhida):

```
'Admin\Controller\User.index',
'Admin\Controller\User.save',
'Admin\Controller\User.delete',
```

Criando as novas visões

O passo final é a criação das duas *views* necessárias, a *module/Admin/view/admin/user/index.phtml*

```
<div class="actions clearfix">
  <div class="btns">
    <a class="btn submit" href="/admin/user/save" title="Criar Usuário">
      Criar Usuário
    </a>
  </div>
</div>
<label class="divisor"><span>Lista de Usuários</span></label>
<table class="datatable">
<thead>
    <tr>
        <th>Nome</th>
        <th>Login</th>
        <th width="120" class="center">Opções</th>
    </tr>
</thead>
<tbody>
<?php foreach($users as $user):?>
    <tr>
        <td><?php echo $this->escapeHtml($user->name);?></td>
        <td><?php echo $this->escapeHtml($user->username);?></td>
        <td class="center">
            <a href="/admin/user/save/id/<?php echo $user->id ;?>"
                title="Editar" class="btn">
              <i class="icon-edit"></i>
            </a>
            <a href="/admin/user/delete/id/<?php echo $user->id;?>"
                rel="confirmation" title="Deseja excluir este registro?"
                class="btn">
              <i class="icon-remove"></i>
            </a>
```

```
32          </td>
33      </tr>
34  <?php endforeach;?>
35  </table>
```

https://gist.github.com/4038458

E a *module/Admin/view/admin/user/save.phtml*:

```
1  <?php
2  echo $this->form()->openTag($form);
3  echo $this->formCollection($form);
4  echo $this->form()->closeTag();
```

Executando os testes

Antes de executarmos os testes novamente, uma observação. Como alteramos a entidade *User* para usarmos o *Doctrine* os testes que usam essa entidade vão parar de funcionar (como o *AuthTest* e o *AuthControllerTest*). Isso é algo esperado, pois fizemos uma mudança grande na estrutura do projeto, incluindo o *ORM*, então precisaremos reescrever os testes para que estes entendam a mudança. Deixo isso como exercício para o leitor.

Enquanto isso podemos executar apenas os testes do novo controlador com o comando:

```
1  phpunit -c module/Admin/tests/phpunit.xml --group=Controller --filter=testU\
2  ser
```

Conforme comentado no início deste capítulo, o *Doctrine* é uma excelente ferramenta e possui uma documentação vasta que deve ser lida com atenção. O objetivo aqui era demonstrar a integração entre o Zend Framework 2 e o *ORM*.

Conclusão

Certamente não consegui aqui englobar todas as funcionalidades do Zend Framework, mas esse não era exatamente o objetivo deste livro. A idéia aqui era apresentar as principais características e como usá-las em uma aplicação comum e espero ter atingido esse modesto objetivo.

Uma das vantagens de ser um e-book é que esse livro pode ser "vivo", coisa que é mais difícil para um livro impresso. Todos os trechos de códigos do PDF possuem links para uma versão online, o que facilita a cópia para o seu editor de programação favorito, mas que principalmente facilita alguma possível correção que venha a acontecer.

Então acompanhe o site oficial deste livro, o http://www.zfnapratica.com.br para acompanhar qualquer melhoria ou correção nos códigos ou textos. E caso tenha alguma sugestão ficaria feliz em recebê-la e publicar no site.

Para entrar em contato comigo a maneira mais fácil é pelo meu site pessoal, o http://eltonminetto.net. Lá você encontra todos os meus contatos, como e-mail, Twitter, etc.

É isso. Happy coding para você!