Conteúdo

Introdução

O Zend Framework 2[1], ou "ZF2" como chamaremos daqui pra diante, é um framework de código aberto para o desenvolvimento de aplicações e serviços web. Ele representa a evolução do Zend Framework 1, versão que teve mais de 15 milhões de downloads.

O Zend Framework tem como principal mantenedora a Zend Technologies, principal empresa por trás da linguagem PHP, mas diversas outras empresas contribuiram no seu desenvolvimento como Google, Microsoft e StrikeIron, para citar algumas.

Considerado um dos frameworks mais robustos para a linguagem PHP, o ZF2 implementa conceitos modernos como Injeção de Dependência e baixo nível de acoplamento, testes e Integração Contínua, suporte a Serviços e Eventos, entre outros.

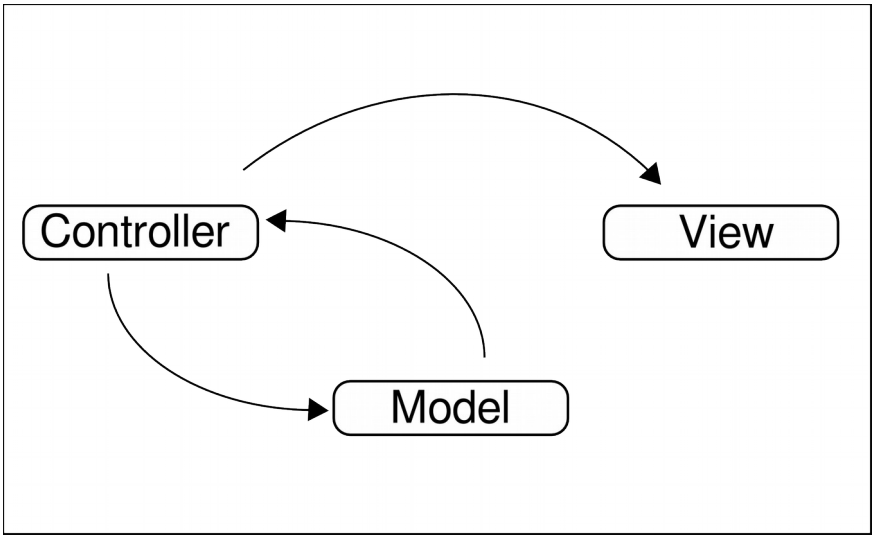
É categorizado como um Framework MVC Full Stack, ou seja, um framework que atende todas as necessidades de desenvolvimento de uma aplicação web e que oferece componentes encapsulados para as tarefas mais comuns.

MVC – Model, View, Controller

MVC é um paradigma de desenvolvimento em camadas, onde procura-se separar as partes (camadas) que compõem uma aplicação. A camada de Model (Modelo) representa os dados, ou seja, frequentemente esta camada representa um banco de dados. A View é a camada de apresentação ou interface e é nela que se encontra a parte visual da aplicação. A Controller, por sua vez, é a camada de processamento, onde se encontra a parte mais pesada de lógica.

A grande vantagem deste paradigma é a separação de código de processamento, dados e código de apresentação, fazendo com que a aplicação se torne mais clara e mais simples de se compreender.

O que um framework MVC faz é implementar esse paradigma, criando a comunicação entre as camadas. No exemplo mais clássico, a Controller se comunica com a Model para, por exemplo, obter dados. A Model devolve estes dados para a Controller que, por sua vez, envia-os para a View para exibição, como mostra a figura abaixo.



Injeção de Dependência e Baixo Acoplamento

Cada componente do ZF2 é desenvolvido com estes dois conceitos em mente. A Injeção de Dependência[2] é um conceito onde, na sua forma mais simples, faz com que ao invés de extender diretamente uma superclasse, a subclasse tenha injetado em seu construtor um objeto desta. Isso faz com que seja mais simples testar as classes, além de permitir que objetos variados (como por exemplo um objeto que extende a classe que define o objeto a ser injetado) sejam passados para os métodos.

A Injeção de Dependência faz com que o ZF2 tenha um nível bem mais flexível de acoplamento onde, por exemplo, seus componentes podem ser utilizados de forma independente do framework como um todo.

Testes e Integração Contínua

O ZF2 tem suas classes testadas através do framework de testes PHPUnit[3] e sua Integração Contínua assegurada através do Travis CI[4]. Isto assegura uma maior qualidade do seu código-fonte, gerando uma confiança maior ao se optar pelo ZF2 como framework para desenvolvimento.

O ZF2 conta atualmente (na data em que este material é escrito) com 84% de seu código-fonte coberto por testes.

Suporte a Serviços e Eventos

O ZF2 traz duas novas funcionalidades para o desenvolvimento web: Serviços e Eventos.

Um Serviço é basicamente um objeto que executa uma lógica complexa de aplicação. O processo de autenticação, por exemplo, pode ser encapsulado como um serviço disponível a aplicação.

Um Evento nada mais é do que uma ação definida por um nome. Ao se definir um evento criamos também um “ouvinte” (*Listener*) que é basicamente o código que reagirá a ocorrência deste evento.

1. https://github.com/zendframework/zf2
2. http://pt.wikipedia.org/wiki/Inje%C3%A7%C3%A3o_de_depend%C3%AAncia
3. https://phpunit.de/
4. http://travis-ci.org/

Pré-Instalação

Pré-requisitos

- PHP 5.3.3+;
- Um servidor web como o Apache, por exemplo;
- Composer ou Pyrus (veja abaixo).

Apresentando o Composer

O Composer[1] é a forma mais moderna e eficiente de gerenciar dependências de um projeto PHP. Utilizando-se de um sistema poderoso, mas simples, de definição de dependências através de um arquivo JSON, o Composer rapidamente tomou o lugar de outras soluções como PEAR e Pyrus. Hoje podemos afirmar sem medo de errar que o Composer se tornou o padrão informal de gerenciamento e instalação de dependências no mundo PHP.

Instalando o Composer

Existem 3 formas de se instalar o Composer:

1. Via cURL:

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

2. Via PHP (desde que a diretiva allow_url_fopen esteja habilitada):

   php -r "readfile('https://getcomposer.org/installer');" | php

3. Ou pela seção Download do próprio site do Composer.

1. https://getcomposer.org/

Instalação

O ZF2 pode ser instalado de duas formas:

1. Por projeto
2. Globalmente

Embora a instalação global tenha as suas vantagens, neste primeiro curso sobre o assunto trataremos da instalação por projeto, mais comum e mais simples.

A Skeleton Application

De forma a facilitar a adoção do ZF2 a Zend desenvolveu o que chamamos de Skeleton Application (ou, literalmente, “Aplicação Esqueleto”). A Skeleton, como chamaremos daqui pra diante, foi desenvolvida tendo como base o tutorial de Rob Allen[1] e tem como objetivo prover um início rápido no desenvolvimento com o ZF2. É baseando-se na Skeleton que desenvolveremos o nosso projeto.

Instalando a Skeleton

Para instalar a Skeleton – que, ressaltamos, será a base para o nosso projeto – utilizaremos o composer:

```
cd /var/www
php composer.phar create-project -sdev --repository-url="https://packages.zendframework.com" zendframework/skeleton-application ./loja
```

Isto fará com que o composer crie o diretório do projeto (“loja”) e já instale, automaticamente como dependência, o ZF2.

Criando o VHost

Para que funcione de forma apropriada, um projeto com o ZF2 necessita de um VHost (Virtual Host, ou "Servidor Virtual"), o que há anos é comum no desenvolvimento de aplicações web. A criação de um VHost depende de dois fatores: Sistema Operacional e Servidor Web.

Veremos como criar um VHost no Sistema Operacional Linux, usando o Servidor Web Apache (para outros Sistemas Operacionais e/ou Servidores Web, a documentação do Zend Framework[2] e uma pesquisa pelo Google podem ajudar):

1. Criar o arquivo de configuração do VHost:

   ```
   cd /etc/httpd/conf.d
   gksudo gedit loja
   ```

2. No arquivo, colocar o seguinte:

   ```
   <VirtualHost *:80>
      ServerName loja
      DocumentRoot /var/www/html/loja/public
      <Directory /var/www/html/loja/public/>
         AllowOverride All
         Order allow,deny
         Allow from all
      </Directory>
   </VirtualHost>
   ```

3. Abrir o arquivo de hosts para configurar a máquina para acessar o VHost como local:

   ```
   gksudo gedit /etc/hosts
   ```

4. Neste arquivo adicionar a linha:

   ```
   127.0.0.1 loja
   ```

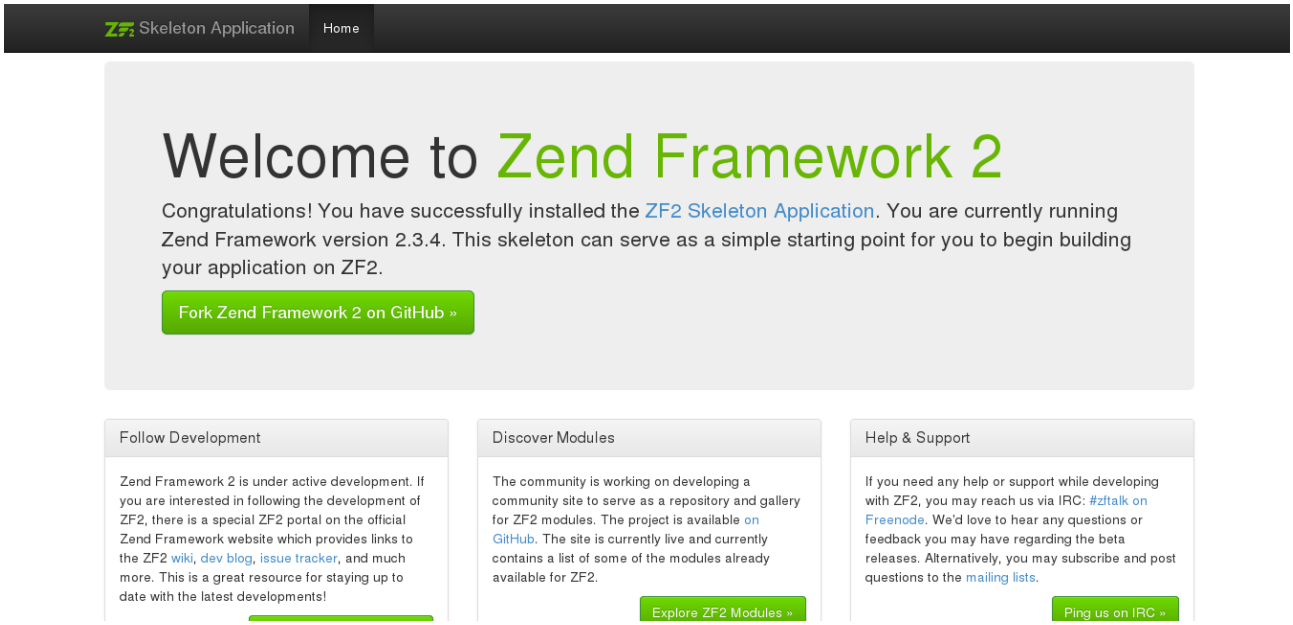
5. Parar e iniciar o Apache:

   ```
   sudo service httpd stop
   sudo service httpd start
   ```

Testando o VHost

Abra o navegador e digite o endereço: http://loja

Se tudo estiver OK, você enxergará uma página como a figura abaixo:



1. http://akrabat.com/
2. http://framework.zend.com/manual/current/en/ref/installation.html

Configurando o Projeto

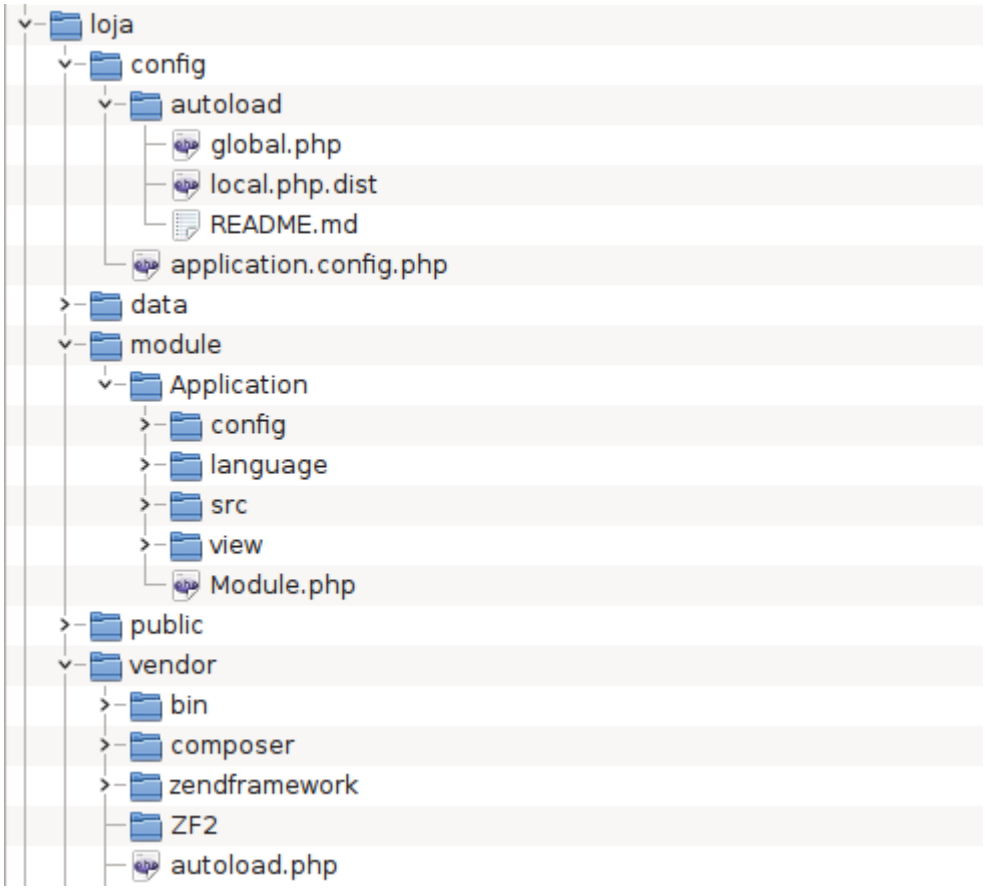
Uma das grandes vantagens do ZF2 sobre diversas outras opções de frameworks PHP é a extrema flexibilidade proporcionada ao desenvolvedor. Da estrutura de diretórios aos nomes de arquivos, praticamente tudo pode ser modificado e customizado de acordo com a necessidade/vontade do desenvolvedor.

Esta flexibilidade vem, evidentemente, com um preço: o projeto precisa ser configurado extensivamente para que o ZF2 compreenda o funcionamento do mesmo.

Para este curso, entretanto, seguiremos o padrão sugerido pela Skeleton.

A Estrutura de Diretórios

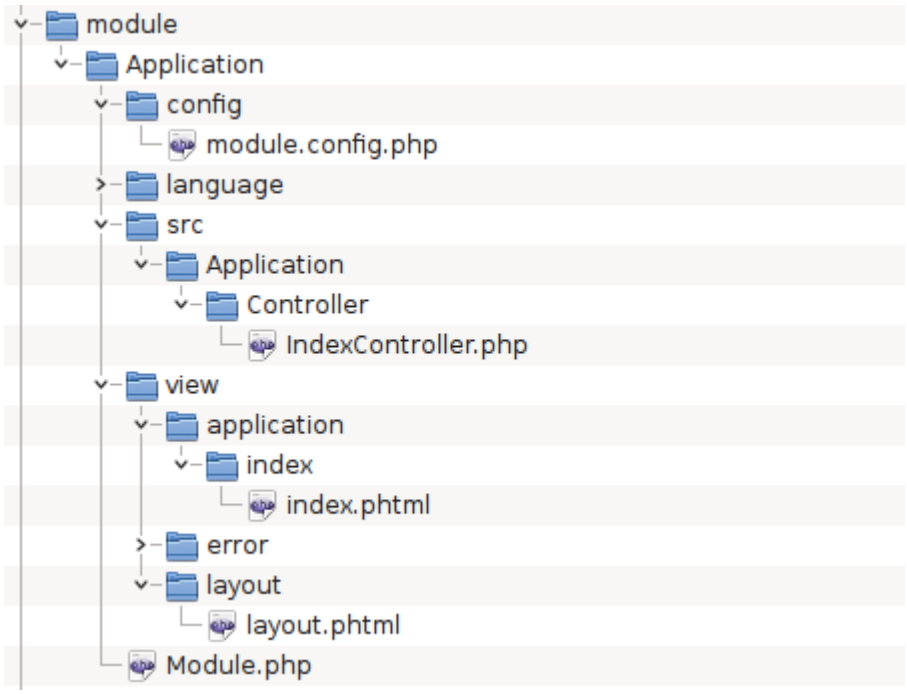
A primeira coisa que precisamos entender é a estrutura de diretórios em um típico projeto baseado na Skeleton.



Embora possa parecer bastante coisa, a Skeleton é, na realidade, bem simples. Vamos detalhar o papel de cada pasta:

- config: É a configuração geral a nível de projeto;
  - application.config.php: configurações “gerais” da aplicação, como por exemplo os módulos que serão utilizados;
  - autoload: contém as configurações de ambientes. Nesta pasta encontram-se os arquivos:
    - global.php: guarda as configurações globais, comuns a todos os ambientes;
    - local.php (que vem, propositalmente como .dist): contém as configurações “locais”, específicas para o ambiente onde a aplicação está sendo executada. Neste arquivo podem ser colocados, por exemplo, senhas de acesso, caminhos (*paths*) específicos, etc...
- data: pasta de dados, ideal para coisas como cache, logs da aplicação, etc...
- module: onde ficam os módulos da aplicação
  - Application: o módulo de exemplo da Skeleton. Normalmente re-utilizamos este módulo para representar a parte “pública” de nossa aplicação. Detalharemos isso melhor em uma imagem específica desta pasta, logo abaixo.
- public: A raiz web da aplicação;
- vendor: A pasta onde ficam os componentes externos (não desenolvidos por você) que a aplicação utiliza, incluindo o próprio ZF2, que fica na pasta zendframework.

Estrutura de Diretórios: O módulo



O ZF2 trabalha em cima do conceito de módulos. Um módulo é algo que possui uma lógica de funcionamento própria e que pode ser compartimentalizado. No caso de nossa loja, a aplicação inteira é um módulo. Futuramente, se necessário, podemos adicionar módulos extras, como por exemplo, um módulo de admnistração da loja.

O ZF2 trabalha com alguns padrões mínimos para o funcionamento do MVC:

- Módulos podem possuir Models, Views e Controllers (além de outros itens, como Layout, Serviços, Eventos, etc...)
- Controllers possuem Actions, que serão relacionadas a rotas (ver o próximo item)
- Cada Controller precisa de uma pasta relacionada (com o mesmo nome da Controller) na camada de view
- Cada Action precisa de um arquivo relacionado (com o mesmo nome da Action) na pasta relacionada a Controller a qual pertence, dentro da camada de view.

Vejamos o significado de cada pasta em uma estrutura típica de um módulo baseado no padrão da Skeleton:

- config: Guarda configurações específicas do módulo, no arquivo module.config.php.
- language: Pasta que guarda os arquivos de tradução, caso nossa aplicação seja multi-idioma.
- src: A palavra vem de “source” (“fonte”) e é a pasta onde se encontra o código-fonte (a lógica) do módulo.
    - Application: o nome do módulo é propositalmente repetido para fins de auto-carregamento baseado em namespaces.
        - Controller: A pasta onde serão salvas as Controllers.
            - IndexController.php: A Controller “Index”
- view: A pasta onde serão salvos os arquivos da camada de view.
    - application: Uma vez mais, o nome do módulo é repetido para viabilizar o auto-carregamento via namespaces.
        - index: Uma pasta referente a views relacionadas a Controller “Index”.
            - index.phtml: A view da ação “index”, da controller “Index”
    - layout: A pasta onde serão salvos os arquivos de layout.
        - layout.phtml: O arquivo de layout do módulo Application.
- Module.php: Arquivo utilizado para definir junto ao framework como o módulo funciona.

Acesso a Banco de Dados

A configuração do acesso ao banco é extremamente fácil e simples. Vamos dividí-la em dois arquivos separados de forma a garantir o sigilo das credenciais de acesso. Este ponto é particularmente importante porque vem de encontro a uma prática de mercado que tem causado problemas há bastante tempo: o fato de que todos os envolvidos em um projeto tem posse das credenciais de acesso aos bancos de dados.

O que faremos é colocar as informações que são independentes de ambiente no arquivo global.php e as credenciais – que deveriam, de qualquer forma, ser diferentes em cada ambiente – no arquivo local.php. Ambos os arquivos estão localizados em loja/config/autoload.

No arquivo global.php nosso array de configuração ficará assim:

```
return array(
    'db' => array(
        'driver' => 'Pdo_Mysql',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
);
```

Apenas os detalhes comuns a todos os ambientes (driver de acesso e comando de inicialização) são definidos.

No arquivo local.php nosso array de configuração ficará assim:

```
return array(
    'db' => array(
        'dsn'    => 'mysql:dbname=loja;host=localhost',
        'user'     => 'root',
        'password' => '',
    ),
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\AdapterServiceFactory',
        ),
    ),
);
```

Desta forma cada ambiente – ou seja, cada desenvolvedor que trabalha no projeto – pode ter suas principais credenciais na sua máquina, tornando o próprio processo de desenvolvimento mais seguro e a contratação de novos desenvolvedores mais tranquila.

A entrada que colocamos no array 'factories' informa ao ZF2 a classe que será usada para gerenciar a conexão com banco de dados, fazendo com que o framework conecte e desconecte quando necessário.

Roteamento

Uma rota, em uma típica aplicação web desenvolvida com ZF2, é algo que faz a ligação entre uma URL, uma Controller e uma Action. O programador possui um altíssimo nível de liberdade na definição de rotas. Pode-se optar, por exemplo, por escrever todo o código-fonte (nomes de módulos, classes e métodos) em inglês, como é comum no mercado, mas possuir rotas em português apontando para este código.

Rotas são definidas na configuração do módulo (module.config.php), através de um array dentro do array de configuração, como demonstramos no exemplo a seguir:

```
'router' => array(
    'routes' => array(
        'home' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/',
                'defaults' => array(
                    'controller' => 'IndexController',
                    'action'     => 'index',
                ),
            ),
        ),
    ),
)
```

Uma rota é definida dentro do array routes, que por sua vez fica posicionado dentro do array router. Toda a configuração da rota é feita, por sua vez, também com um array.

Uma rota possui, basicamente, 3 definições ou índices deste array: O nome da rota (neste caso, 'home'), o tipo da rota (neste caso, 'Zend\Mvc\Router\Http\Literal' ou apenas 'Literal') e opções da rota ('options').

Dentre estas opções duas são vitais para o funcionamento da rota: 'route', que define a URL (neste caso '/', ou seja, quando é chamado http://loja/) e 'defaults', que diz para qual Controller e qual Action a rota aponta.

Criando uma rota

Agora que sabemos como funciona o roteamento no ZF2, vamos criar nossa primeira rota. No arquivo module.config.php, adicionaremos uma rota para a página de produtos de nossa loja:

```
'produtos' => array(
    'type' => 'Zend\Mvc\Router\Http\Literal',
    'options' => array(
        'route' => '/produtos',
        'defaults' => array(
            'controller' => 'ProdutosController',
            'action' => 'index'
        )
    )
)
```

Este array configura nossa nova rota. Damos um nome a ela('produtos') e definimos as suas configurações. Evidentemente não basta criar uma rota. Precisamos agora criar a Controller e a Action que responderão a esta rota, além de uma view para exibir o conteúdo.

Mais sobre rotas: Segmentos

O ZF2 oferece uma grande variedade de tipos de rotas para atender as mais diversas necessidades de roteamento em uma aplicação[1]. Dentre estes, o mais importante e comum é o tipo de Segmentos. Este tipo permite a definição de rotas variáveis e que portanto tornam o processo de configuração do projeto menos extenso, além de permitir o recebimento de dados via query string.

Vejamos um exemplo de configuração de uma rota com segmentos:

```
'produto-visualizar' => array(
    'type'    => 'Zend\Mvc\Router\Http\Segment',
    'options' => array(
        'route'       => '/produto/:id',
        'defaults' => array(
            'controller' => 'ProdutosController',
            'action'     => 'visualizar',
        ),
    ),
),
```

A diferença desta rota para a que criamos anteriormente – além da óbvia diferença de tipo – é a presença de um segmento, representado pelo ":id". Um segmento pode ser entendido como uma espécie de "variável de rota", ou seja, estamos definindo que esta rota irá receber um "id" variável, como por exemplo:

http://loja/produto/1639

Mais sobre rotas: Validação

Rotas com segmentos podem se tornar possíveis vetores de ataques à aplicação. Um usuário mal-intencionado poderia, por exemplo, tentar passar um dado que causasse uma injeção de SQL:

http://loja/produto/1+or+1=1

Solucionar este problema é extremamente simples com o ZF2. Simplesmente definimos constraints ("limitações") para o segmento, dentro das opções da rota. Nossa rota de visualização ficará então assim:

```
'produto-visualizar' => array(
   'type' => 'Zend\Mvc\Router\Http\Segment',
   'options' => array(
      'route'    => '/produto/:id',
      'constraints' => array(
         'id' => '[0-9]+',
      ),
      'defaults' => array(
         'controller'   => 'ProdutosController',
         'action'       => 'visualizar',
         'id'           => 0,
      ),
   ),
),
```

Constraints são Expressões Regulares que definem o formato de um segmento, impedindo a sua violação. Nesta expressão, estamos definindo que "id" é um número composto de um ou mais dígitos. Além disso, definimos um valor default para "id", ou seja, caso não nos seja informado nenhum id nossa aplicação considerará o valor do segmento como zero.

1. http://framework.zend.com/manual/current/en/modules/zend.mvc.routing.html

Controllers

Controllers, na ótica da Skeleton, possuem três características em comum:

1. Seus arquivos são salvos na pasta module/nome_do_módulo/src/nome_do_módulo/Controller;
2. Seu arquivo e nome de classe terminam pela palavra “Controller”;
3. Extendem a classe Zend\Mvc\Controller\AbstractActionController.

Criando um Controller

Na pasta correta, como colocamos acima, criaremos um arquivo chamado ProdutosController.php. Neste arquivo colocaremos o seguinte código:

```
<?php
namespace Application\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class ProdutosController extends AbstractActionController
{
   public function indexAction()
   {
      return new ViewModel();
   }
}
```

Este código é bem simples. O que ele faz (pela ordem das linhas) é exatamente o que segue:

1. Definimos que essa classe pertence ao namespace Application\Controller. Isso auxiliará o autoloader do Zend Framework a encontrar nossa classe.
2. Definimos as classes que serão usadas em nossa controller: AbstractActionController (que fará com que nossa controller seja compreendida como uma Action Controller) e ViewModel (que fará a comunicação Controller → View);
3. Definimos que nossa classe extende a classe AbstractActionController do ZF2;
4. Deifnimos um método que – por enquanto – apenas libera a View para exibição.

Liberando a Controller para acesso

Uma das preocupações ao desenvolver o ZF2 foi de permitir que o desenvolvedor configure a aplicação extensamente. Esta configuração nos dá diversas vantagens, como por exemplo a possiblidade de criarmos nossos próprios padrões de aplicação. Uma vantagem que não é óbvia, entretanto, é que a configuração extensa da aplicação também permite o desenvolvimento de uma aplicação mais segura.

Imagine a seguinte situação: você precisa desenvolver uma nova controller, mas ela não pode, de forma alguma, ser acessada até que esteja pronta. O ZF2 resolve essa situação hipotética fazendo com que as Controllers só sejam acessíveis se forem configuradas como tal. Isto é feito no arquivo de configuração do módulo (module/nome_do_módulo/config/module.config.php), através de um array chamado, obviamente, “controllers”:

```
'controllers' => array(
    'invokables' => array(
        'IndexController' => 'Application\Controller\IndexController'
    ),
),
```

O que precisamos fazer é, simplesmente, incluir nossa controller entre as “invokables” (ou “invocáveis”):

```
'controllers' => array(
    'invokables' => array(
        'IndexController' => 'Application\Controller\IndexController'
        'ProdutosController' => 'Application\Controller\ProdutosController'
    ),
),
```

Observe que o índice do array invokables é para onde o índice “controller” do array “defaults” da rota aponta.

Processando Dados Externos

Segmentos podem ser utilizados de diversas formas, mas uma das forma mais comum é para a transmissão de dados via query string (como é o caso de nossa rota 'produto-visualizar'). O fato de que o segmento é parte da rota torna extremamente simples o recebimento do dado passado.

Ao extender a AbstractActionController, herdamos uma série de métodos que podemos chamar de “utilitários” para trabalhar. Ao trabalharmos com segmentos na verdade estamos definindo, de uma certa forma, parâmetros de rota que são armazenados em um objeto. Este objeto pode ser recuperado por um método herdado chamado 'params' e, através de um método linkado, obtemos o(s) parâmetro(s) de rota.

Vamos agora, fazer o recebimento do segmento 'id' em uma action 'visualizar' de nossa controller de produtos:

```
public function visualizarAction()
{
   $id = $this->params()->fromRoute('id');
}
```

O método 'fromRoute' especifica que o parâmetro que estamos buscando faz parte da rota, e que no contexto da rota ele se chama 'id'. Simples assim.

Dados de formulários

Quem desenvolve aplicações web sabe que nem todo o dado externo é passado como segmento. Na verdade a maioria dos casos não envolve (ou pelo menos não necessariamente) segmentos de rotas. Dados enviados por formulários, por exemplo, são enviados através da própria requisição HTTP.

Uma das características mais interessantes do ZF2 é a sua preocupação com um dos principais conceitos de Orientação a Objetos: a generalização. É através deta generalização que o ZF2 torna extremamente simples a recuperação de dados enviados pela requisição.

A requisição, no contexto do ZF2, é um objeto que já nos é passado “automaticamente” através da herança da AbstractActionController. Tudo o que precisamos fazer, portanto, é trabalharmos com este objeto para que possamos recuperar os dados desejados.

Vejamos um exemplo simples de uso do objeto da requisição para recuperar dados enviados. No próximo capítulo expandiremos este exemplo.

```
public function cadastrarAction()
{
   $requisicao = $this->getRequest();
   $dados = $requisicao->getPost();

   return new ViewModel();
}
```

O método 'getPost' do objeto da requisição recupera os dados enviados por POST e os retorna em um array associativo, como o $_POST ao qual já estamos acostumados ao trabalhar com PHP.

Views

A camada de View é, provavelmente, a camada mais simples para trabalharmos com o ZF2. Isto se deve em parte ao fato de que uma view, tecnicamente falando, não precisa nem mesmo conter código PHP.

Criamos, até este momento, ações em uma controller. Sem a camada de View não nos é possível testar estas ações, portanto agora veremos alguns exemplos simples de utillização desta camada com o ZF2.

Arquivos de views, no contexto da Skeleton, possuem duas características comuns:

1. São salvos em module/nome_do_modulo/view/nome_da_controller/
2. O arquivo da view possui o nome da action e a extensão .phtml

Vamos criar uma view para nossa action 'visualizar', da controller 'produtos', ou seja, salvaremos um arquivo chamado visualizar.phtml dentro da pasta module/Application/view/produtos/

A maneira mais simples (simplória, até) para testarmos uma rota → controller → action → view é simplesmente criarmos um arquivo em branco ou mesmo apenas contendo código HTML:

```
<div>Esta é a view da action 'visualizar' da controller 'produtos'.</div>
```

Embora útil para testarmos as partes mais simples da aplicação, não possui muita utilidade, certo? Vamos mudar isso:

Comunicação Controller → View

A comunicação entre as duas camadas é algo muito simples de se alcançar com o ZF2. Tudo o que precisamos é fazer com que a controller retorne um array dentro do objeto ViewModel.

Digamos que desejamos definir um título para nossa view e retornar o ID do produto que está sendo visualizado:

```
public function visualizar()
{
    $id = $this->params()->fromRoute('id');

    return new ViewModel(array(
        'titulo' => 'Visualizar Produto',
        'id'     => $id,
    ));
}
```

Isto faz com que os índices definidos neste array se tornem automaticamente variáveis disponíveis para a view:

```
<h1><?=$titulo;?></h1>
<div>Esta é a view da action 'visualizar' da controller 'produtos'.</div>
Visualizando o produto de ID: <?=$id;?>
```

Formulários

Formulários são, basicamente, uma estrutura HTML para entrada de dados. Naturalmente, portanto, o ZF2 não tem nada a ver com isso, certo? Não é tão simples assim.

Embora a geração de formulários via código PHP seja um assunto relativamente polêmico, o ZF2 justifica a sua utilização para gerar formulários devido a facilidade de se realizar tarefas comumente maçantes ou extremamente complexas quando adotamos o componente Zend\Form.

Ao utilizarmos o ZF2 para gerar um formulário, criamos uma classe para este. Devido ao mecanismo de autoload do ZF2 e a padronização de processos de namespaces e autoloading promovidos pela PSR-0 e pela PSR-4, podemos salvar nosso formulário praticamente onde quisermos. Como o formulário é um componente importante do módulo, contudo, salvaremos ele em uma pasta 'Form' (com a adição de algumas pastas, como veremos em seguida) dentro da pasta module/nome_do_modulo/src/ nome_do_modulo.

Criando um formulário

Para mantermos nossa aplicação o mais organizada possível – embora seja sempre importante lembrar que isso é apenas uma recomendação – criaremos uma pasta para a entidade e daremos ao nosso formulário o nome da ação na qual ele toma parte.

Sendo assim, ele será salvo em module/Application/src/Application/Form/Produto e seu arquivo terá o nome de Cadastrar.php

```php
<?php
namespace Application\Form\Produto;

use Zend\Form\Element;
use Zend\Form\Form;

class Cadastrar extends Form
{
   public function __construct($name = null)
   {
      parent::__construct($name);

      $nome = new Element\Text('nome');
      $nome->setLabel('Nome do Produto: ');


      $botao = new Element\Submit('cadastrar');
      $botao->setValue('Cadastrar');

      $this->add($nome);
      $this->add($botao);
   }
}
```

Como podemos perceber o funcionamento de uma classe de formulário é tão simples que é praticamente intuitivo: tudo o que precisamos fazer é:

1. Extender a classe Zend\Form\Form
2. instanciá-la
3. Criar nossos elementos usando as classes do namespace Zend\Form\Element[1]. Observe que a string passada para o construtor será o valor do atributo HTML name.
4. Adicionar os elementos criados ao objeto do form.

A partir da criação de nosso formulário, agora precisamos instruir a controller a utilizá-lo e a view a exibí-lo.

Na controller a primeira coisa que faremos será “usar” nosso form:

```php
use Application\Form\Produto\Cadastrar as FormCadastro;
```

E depois alterar a action para utilizá-lo.

```
public function cadastrarAction()
{
    $requisicao = $this->getRequest();
    $form = new FormCadastro();

    if ($requisicao->isPost()) {
        $dados = $requisicao->getPost();
    }

    return new ViewModel(array(
        'form' => $form
    ));
}
```

As mudanças são interessantes para esclarecermos:

1. Observe que o formulário é criado independente do método de envio da requisição. Isso é proposital porque, em caso de erros de envio, podemos querer re-exibir o formulário para que o usuário possa corrigí-lo.
2. O método 'isPost' do objeto da requisição, obviamente, retorna um booleano para informar se esta foi enviada por POST.
3. Incluímos o form na ViewModel retornada de forma a poder exibí-lo na view.

Na camada de view o que faremos é, simplesmente, exibir o formulário. Nossa view 'cadastrar.phtml' ficará assim:

```
<?php
$form->setAttribute('action', $this->url('produto-cadastrar', array('action' => 'cadastrar')));
$form->prepare();
?>
<?=$this->form()->openTag($form);?>
<?=$this->formRow($form->get('nome'));?><br>
<?=$this->formSubmit($form->get('cadastrar'));?>
<?=$this->form()->closeTag();?>
```

Models

Models representam os dados de uma aplicação. O mercado se habitou, ao falar de dados, com a sigla C.R.U.D. (Create, Read, Update, Delete), representando as quatro “operações” básicas a se realizar com um dado/registro: Criar (Inserir – Insert), Ler (Obter –

Select), Atualizar (Modificar – Update) e Excluir (Deletar – Delete).

Vejamos de que forma o ZF2 trabalha com estas operações.

Setup da Model

A Model é uma camada mais complexa do que as duas anteriores, e portanto precisa de uma certa preparação. Cada entidade, na ótica da Skeleton, ganha dois arquivos: Um Model (uma representação da entidade em forma de classe) e um TableGateway, uma classe que possui as interações e processamentos necessários para a realização das quatro operações.

Este setup permite que utilizemos a Model diretamente via ServiceManager: no arquivo module/nome_do_modulo/Module.php fazemos com que o ServiceManager carregue as classes de model diretamente para nós.

Primeiramente utlizamos a instrução 'use' para facilitar o uso da Model:

```
use Application\Model\Produto;
use Application\Model\ProdutoTable;
use Zend\Db\TableGateway\TableGateway;
```

Depois fazemos o código que carregará a camada, através de um método denominado 'getServiceConfig':

```
public function getServiceConfig()
{
    return array(
        'factories' => array(
            'ProdutoTable' =>  function($sm) {
                        $tableGateway = $sm->get('ProdutoTableGateway');
                        $table = new ProdutoTable($tableGateway);
                        return $table;
            },
            'ProdutoTableGateway' => function ($sm) {
                        $dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
                        $resultSetPrototype = new ResultSet();
                        $resultSetPrototype->setArrayObjectPrototype(new Produto());
                        return new TableGateway('produto', $dbAdapter, null,
                            $resultSetPrototype);
            },
        )

    );
}
```

Este método fará com que a camada de Model esteja facilmente disponível para a Controller. Vejamos agora como criar a classe Model e a classe TableGateway de nossa entidade.

Criando a camada Model

Vamos iniciar pela Model, mais simples. Ela será salva (como vimos nos namespaces acima) em module/nome_do_modulo/src/nome_do_modulo/Model e seu nome será o nome da entidade com inicial maiúscula.

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

 class Produto
{
    public $id;
    public $nome;

    public function exchangeArray($data)
    {
        $this->id     = (!empty($data['id'])) ? $data['id'] : null;
        $this->nome = (!empty($data['nome'])) ? $data['nome'] : null;
    }


    public function getArrayCopy()
    {
        return get_object_vars($this);
    }
}
```

A classe Model é bem simples: ela nada mais é do que uma representação da entidade no estilo ORM (Object Relational Mapping, ou Mapeamento Objeto-Relacional) e um método que popula o objeto a partir de um array de dados.

A classe TableGateway é bem mais extensa, pois conterá todos os métodos que trabalham com a entidade (ou seja, o C.R.U.D. em si). Por conta disso, vamos explorar ela em partes, comentando quando for relevante. Ela será salva no mesmo diretório, mas seu nome será o nome da entidade com a inicial maiúscula acrescido da palavra Table.

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

use Zend\Db\TableGateway\TableGateway;

class ProdutoTable
{
    protected $tableGateway;

    public function __construct(TableGateway $tableGateway)
    {
        $this->tableGateway = $tableGateway;
    }

    public function fetchAll()
    {
        $resultSet = $this->tableGateway->select();
        return $resultSet;
    }
}
```

O método construtor recebe, através de injeção de dependência, a tablegateway do ZF2. Na verdade é esta classe que, internamente, tratará de “traduzir” o que estamos processando em PHP para o SQL nativo do SGBD.

Ao contrário do que normalmente acontece quando trabalhamos com bancos de dados, o método de seleção de dados é um dos mais simples. É este método, que a Skeleton chama de “fetchAll” (mas que poderíamos chamar como quiséssemos) que faz a seleção de todos os registros de uma tabela. O que o método faz é simplesmente executar o método 'select' da TableGateway do ZF2 e retornar o resultado, apenas isso.

Ao trabalharmos com os métodos seguintes, você percerberá que isso é comum, pois faz com que os dois parágrafos acima façam sentido: o que nossa TableGateway faz, em última instância, é apenas executar um método encapsulado pelo ZF2.

Vejamos o método de exclusão de registro:

```php
    public function deleteProduto($id)
    {
        $this->tableGateway->delete(array('id' => (int) $id));
    }
```

Novamente observe a simplicidade do método: recebemos um ID, e executamos um método da TableGateway do ZF2, passando este ID como parâmetro.

O próximo realiza, sozinho, tanto a atualização quanto inserção:

```
public function saveProduto(Produto $produto)
{
    $data = array(
        'nome'  => $produto->nome,
    );

    $id = (int) $produto->id;

    if ($id == 0) {
        $this->tableGateway->insert($data);
    } else {
        if ($this->getProduto($id)) {
            $this->tableGateway->update($data, array('id' => $id));
        } else {
            throw new \Exception('Produto não existe');
        }
    }
}
```

O conceito deste método é simples: recebe-se uma instância do Model do produto como parâmetro, e procura-se o valor da propriedade 'id' deste objeto. Caso ele seja null (zero, ao ser explicitamente convertido para inteiro) fazemos a inserção. Caso contrário procuramos por este registro no banco (veja a seguir) e se ele de fato existir atualizamos.

Vejamos o método final de nosso C.R.U.D.: o que seleciona um registro apenas, baseando-se em seu ID:

```
public function getProduto($id)
{
    $id  = (int) $id;
    $rowset = $this->tableGateway->select(array('id' => $id));
    $row = $rowset->current();

    if (!$row) {
        throw new \Exception("O registro de ID $id não foi encontrado");
    }

    return $row;
}
```

Este método roda o mesmo método da TableGateway do ZF2 ('select'), mas passando um array que será usado como cláusula. Como sabemos, SGBDs retornam um conjunto de registros mesmo quando retornam apenas um registro, portanto precisamos do método 'current' para obter os dados.

Queries customizadas

Evidentemente existem diversos casos não previstos na Skeleton – afinal, ela se destina a ser um rápido exemplo de MVC com o ZF2. Para todas as necessidades de bancos de dados não previstos, podemos construir queries através da utilização do componente Zend\Db\Sql, além de outros componentes para nos auxiliar:

```
use  Zend\Db\Sql;
use  Zend\Db\ResultSet\ResultSet;
use Zend\ServiceManager\ServiceLocatorAwareInterface;
use Zend\ServiceManager\ServiceLocatorInterface;
```

Além da inclusão destes componentes, precisamos especificar que nossa classe TableGateway implementa a interface ServiceLocatorInterface e criar alguns métodos auxiliares:

```
public function setServiceLocator(ServiceLocatorInterface $serviceLocator)
{
    $this->serviceLocator = $serviceLocator;
}

public function getServiceLocator()
{
    return $this→serviceLocator;
}

public function setAdapter()
{
    $this->dbAdapter = $this→getServiceLocator()→get('Zend\Db\Adapter\Adapter');
}

public function getAdapter()
{
    if (!$this->dbAdapter) {
        $this→setAdapter();
    }

    return $this→dbAdapter;
}
```

Vejamos, como exemplo, uma construção equivalente a seguinte query SQL:

```
SELECT * FROM produto WHERE nome = 'Notebook 14 polegadas';
```

Utilizaremos o método 'select' do componente Zend\Db\Sql:

```
$sql = new Sql($this->getAdapter());
$select = $sql->select();
$select->from('produto');
$select->where(array('nome' => 'Notebook 14 polegadas'));

$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
```

O Zend\Db\Sql utiliza, normalmente, o padrão String SQL → Preparar → Executar, ou o que chamamos de “Prepared Statements”, o que incrementa a performance e segurança de operações no banco.

Observer-se que os métodos relacionados a construção da query são extremamente intuitivos: 'from', 'where', etc... Portanto não é difícil presumirmos – corretamente – a existência de outros métodos, como 'order', 'like', 'join', entre outros.

Unindo as Camadas: M + V + C

Já vimos a união das camadas de View e Controller, portanto o que nos falta é vermos como utilizar a camada de Model em conjunto com a Controller.

Preparação

Antes de mais nada nossa controller precisa estar “ciente” da existência da Model. Isso é basicamente feito adicionando-se uma propriedade para armazenar a model e simplesmente implementando-se um método que usa o ServiceManager para carregá-la:

```
public function getProdutoTable()
{
    if (!$this->produtoTable) {
        $sm = $this->getServiceLocator();
        $this->produtoTable = $sm->get('ProdutoTable');
    }

    return $this->produtoTable;
}
```

Seleção

Nossa action index (cuja idéia é listar todos os produtos) em nossa controller de produto ficará da seguinte forma:

```
public function indexAction()
{
    return new ViewModel(array(
        'produtos' => $this->getProdutoTable()->fetchAll(),
    ));
}
```

É em exemplos como esse que percebemos como o extenso trabalho realizado com a Model vale a pena: o método na controller para o processamento se torna incrivelmente enxuto.

E a View? O que precisamos lembrar é o que o método 'fetchAll' da Model nos retorna, pois a Controller está simplesmente servindo de ponte entre as outras duas camadas. A resposta é um objeto que contém cada registro como um objeto Model, portanto na view simplesmente faríamos algo como, por exemplo:

```
foreach ($produtos as $produto) {
    echo $produto->nome . '<br>';
}
```

Seleção de um registro

Na controller:

```
public function visualizarAction()
{
   $id = $this->params()->fromRoute('id');

   return new ViewModel(array(
      'produto' => $this->getProdutoTable()->getProduto($id),
   ));
}
```

Na View o que teremos que fazer é simplesmente exibir as propriedades do objeto Model que desejarmos.

Cadastro (Inserção)

Na Controller:

```
public function cadastrarAction()
{
   $requisicao = $this->getRequest();
   $form = new FormCadastro();


   if ($requisicao->isPost()) {
      $form->setData($requisicao->getPost());

      if ($form->isValid()) {
         $produto = new Application\Model\Produto();
         $produto->exchangeArray($form->getData());
         $this→getProdutoTable()→saveProduto($produto);
      }
   }

   return new ViewModel(array(
      'form' => $form
   ));
}
```

Alteração

Na Controller:

```
public function alterarAction()
{
    $id = (int) $this->params()->fromRoute('id');
    $produto = $this->getProdutoTable()->getProduto($id);

    $form = new FormCadastro();
    $form->bind($produto);
    $form->get('cadastrar')->setAttribute('value', 'Alterar');

    $requisicao = $this->getRequest();

    if ($requisicao->isPost()) {
        $form->setData($requisicao->getPost());

        if ($form->isValid()) {
            $this->getProdutoTable()->saveProduto($produto);
        }
    }

    return new ViewModel(array(
        'form' => $form
    ));
}
```

Duas partes desse código, em particular, merecem nossa atenção, já que o restante deve a esta altura estar bem compreendido:

O método 'bind' do objeto form nos permite 'amarrar' um objeto Model ao formulário, fazendo com que os dados originais do registro sejam automaticamente populados no formulário. Isso facilita enormemente um dos trabalhos mais maçantes no desenvolvimento de um formulário de edição.

Além disso, alteramos, em tempo de execução, o texto do botão de submit, através do método 'setAttribute'. Este método evidentemente pode ser usado com qualquer elemento de formulário o que, embora incomum, abre possibilidades interessantes.

Conclusão

O ZF2 representa, de muitas formas, o que há de mais moderno em desenvolvimento PHP atualmente. O conteúdo a respeito dele, e portanto o aprendizado e o aperfeiçoamento de seu uso, é extremamente extenso.

Siga estudando, pesquisando, se aprimorando. Vá além da Skeleton. Encontre os “defeitos” que a Skeleton possui, como ela poderia ser melhorada. Acima de tudo, olhe para ela pela ótica de Orientação a Objetos.

Melhore o que pode ser melhorado, generalize o que pode ser generalizado. Seja criativo, mas acima de tudo seja eficiente.