Padrões, PEAR e Frameworks PHP

Professor: FLÁVIO GOMES DA SILVA LISBOA (FGSL)

AULA 4 – Camada de Apresentação

Templates de Página e Formulários Dinâmicos

Plano de Aulas

Dia	Conteúdo
1	Motivação para o uso de frameworks. Instalação e uso do Eclipse com plugin PDT. Padrão de Projeto MVC. Apresentação do Zend Framework. Projeto Mínimo. Padrões de Projeto Singleton, Controller Front e Controller Page. Controle de Erros.
2	Componente de acesso ao banco. Mapeamento Objeto-Relacional. Abstração da camada do banco X Uso de funções específicas. Encapsulamento da sessão como objeto. Padrões de Projeto Factory, Gateway, Iterator e Active Record.
3	Implementação de código seguro com componentes do framework. Filtros e Validadores. Listas de Controle de Acesso. Autenticação. Segurança no acesso ao banco de dados.
4	Separação da aplicação em módulos. Uso de templates e subtemplates de página. Criação de formulários dinâmicos.
5	Encapsulamento de componentes de terceiros (PEAR, Smarty). Criação de novos componentes.

"Frameworks só atrapalham"

Parece contraditório dizer que *frameworks* vêm para auxiliar o desenvolvimento, quando parece que a utilização dos mesmos é mais complicada que fazer as coisas da maneira tradicional.

Na verdade, a impressão inicial é que é muito mais difícil fazer as mesmas coisas quando se adota um *framework*.

Pensando grande

Podemos usar uma parábola para tentar convencê-lo do propósito dos *frameworks*.

Programar sem *frameworks* é como construir uma casa edificada sobre a areia. Na primeira tempestade, a casa cai.

Como diria Jack Nicholson, “pense no futuro”!

Aplicação dividida em módulos

"Dividir para conquistar" poderia ser uma frase de Sun Tzu. Ele disse algo parecido, quando falava de táticas de guerra. Na verdade ele disse: "se o inimigo está em maior número, divida suas forças".

Às vezes o inimigo não precisa necessariamente ser um exército. De qualquer forma, uma coisa é certa: ele é um **problema**.

Aplicação dividida em módulos

René Descartes, na obra *Discurso sobre o Método*, enuncia alguns passos para resolver qualquer tipo de problema. Um deles é dividir o problema em partes menores. Isso se chama **análise**.

Análise de sistemas, portanto, significa dividir sistemas em partes menores, os **subsistemas**.

Aplicação dividida em módulos

Sistemas de informação crescem tanto e tão rapidamente quanto ervas daninhas, vírus, coelhos na Austrália, nuvens de gafanhotos ou outras pragas.

Edsger Djikstra disse que nossa mente não tem capacidade para compreender e acompanhar a complexidade dos programas de computador.

Aplicação dividida em módulos

Martin Fowler, por outro lado, disse que "qualquer um faz um programa que um computador entende. Bons programadores fazem programas que humanos podem entender".

Programas não são códigos secretos de sociedades secretas!

Aplicação dividida em módulos

A experiência mostra que explanações longas só conseguem um efeito: provocar o sono de seus ouvintes. O mesmo ocorre com textos longos.

Códigos-fonte de programas não fogem à regra. Conforme eles se tornam maiores, ficam cada vez mais entediantes. E mais difíceis de entender.

Aplicação dividida em módulos

Por isso é necessário dividir um programa em partes menores, que sejam curtas o suficiente para serem compreendidas isoladamente.

Por outro lado, a arquitetura da aplicação deve ser simples o suficiente para que compreendamos como as peças se encaixam.

Aplicação dividida em módulos

O assunto a ser tratado aqui brevemente, antes de falarmos da camada de apresentação, só faz sentido realmente para aplicações grandes.

Quando a aplicação cresce a tal ponto que parece haver várias “sub-aplicações”, é a hora de dividir a aplicação em módulos.

Roteamento da aplicação

O Zend Framework facilita a criação de dois tipos de roteamento:

- application/controllerpage/action
- application/module/controllerpage/action

Roteamento da aplicação

O segundo tipo é implementado da seguinte forma:

Ao invés de configurar **um** diretório de **controladores de página** com o método *setControllerDirectory()*, configuramos um diretório de **módulos** com o método *addModuleDirectory()*.

Roteamento da aplicação

Cada módulo terá um diretório contendo uma estrutura MVC.

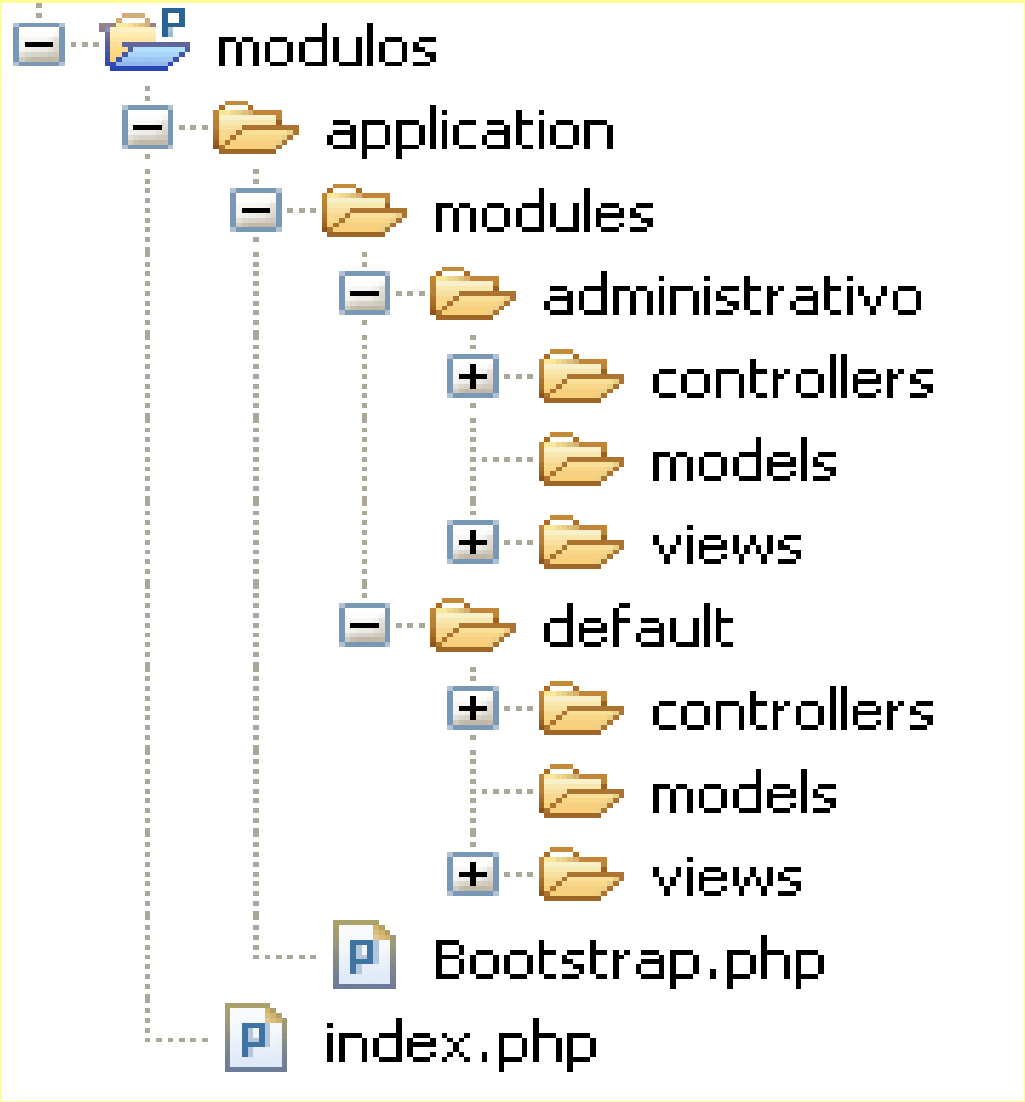
Quando nenhum módulo é especificado na URL, o ZF automaticamente encaminha a requisição para o módulo com o nome de **default**.

Roteamento da aplicação

Exceto para o módulo *default*, as classes herdeiras de Zend_Controller_Action devem ser prefixadas com o nome do módulo que as contém. Mas o nome do arquivo segue o mesmo padrão já visto.

Assim, o arquivo IndexController do módulo administrativo contém a classe *Administrativo_IndexController*.

Estrutura de uma aplicação ZF modular



Camada de Apresentação

CHEGA DE MÓDULOS! QUERO VER A CAMADA DE APRESENTAÇÃO!

Camada de Apresentação

Na primeira aula vimos a camada de controle. Na segunda, vimos a camada de modelo. Agora veremos a camada de apresentação, aquilo que o usuário pensa ser a aplicação.

Na verdade, já trabalhamos com *templates* de páginas, da forma mais simples.

Template padrão

Por padrão, todo método de um objeto Zend_Controller_Action que trata uma requisição (termina com o sufixo *Action*) é sucedido pela renderização de um template cujo nome é o mesmo do método e cuja extensão é **phtml**.

O *template* de página é procurado por padrão dentro de **views/scripts**, em um diretório com o nome do controlador.

Template padrão

Por exemplo, o método *indexAction()* de **Index**Controller, é sucedido pela renderização do arquivo **index**.phtml, dentro de views/scripts/**index**.

Nesse modo, não é preciso dar um único comando para que a renderização seja feita. E ela só não será feita se ocorrer um redirecionamento para outro método ou interrupção com o comando **exit**.

Passando dados para o template

FORMA DIRETA:

```
$this->view->mensagem = 'Feito por mim!';
```

FORMA P**OO**LITICAMENTE CORRETA

```
$this->view->assign('mensagem','Feito por mim!');
```

Exibindo os dados no template

```
<?=$this->mensagem?>
```

A palavra **$this** aqui se refere ao objeto Zend_View incorporado à instância de Zend_Controller_Action.

Exibindo um outro template

O Controller Page tem que usar outra instância de **Zend_View:**

```
    $view = new Zend_View();
    $view-
>setBasePath('./application/views');
    $view->assign('mensagem','Feito por
mim!');
    $view->render('outro.phtml');
    echo $view->render('outro.phtml');
    exit;
```

Sem **exit**, haverá dupla renderização!

Exibindo um template dentro de outro

O Controller Page tem que usar outra instância de **Zend_View:**

```
    $view = new Zend_View();
    $view->setBasePath('./application/views');
    $view->assign('mensagem','Feito por mim!');
    $this->_response->setBody($view->render('outro.phtml'));
```

Não precisa de **exit!**

Garantindo que a saída vai ser a desejada

Configure a função que vai tratar a saída e a codificação utilizada. Depois, é só usar o método *escape()*.

```
    $view = new Zend_View();
    $view-
>setBasePath('./application/views');
    $view->setEscape('htmlentities');
    $view->setEncoding('UTF-8');
    $mensagem = $view->escape('Açafrão');
    $view->assign('mensagem',$mensagem);
```

Formulários Dinâmicos

Zend Framework permite a construção de formulários HTML em tempo de execução.

Zend_Form provê uma interface onde os elementos HTML podem ser tratados como objetos reutilizáveis.

Definindo formulários

Um formulário é *definido* com a criação e configuração de um objeto Zend_Form():

```
$form = new Zend_Form("login");
$form->setAction('index/login');
$form->setMethod('post');
```

A instância de Zend_Form encerrará o conteúdo do texto HTML referente a um formulário.

Elementos de um Formulário

Os elementos HTML de um formulário são criados com o método *addElement()*.

Esse método recebe como parâmetro um objeto de uma das classes filhas de Zend_Form.

Cada chamada de *addElement()* cria um conteúdo HTML no interior das tags `<form></form>`.

Caixas de texto

```
$text = new
Zend_Form_Element_Text('username');
$text->setLabel('Usuário');
$text->setAttrib('id','username');
$text->setRequired(true);
$text->addDecorator('HtmlTag');
$form->addElement($text);
```

Caixas de texto

O método *setLabel()* cria o rótulo para o elemento HTML.

Qualquer atributo pode ser definido com *setAttrib()*.

Para determinar que o preenchimento do campo é obrigatório, usamos *setRequired()*.

O método *addDecorator()* recebe como parâmetro um objeto que define o *layout* e as funcionalidades disponíveis para o elemento.

Caixas de senha

```
    $text = new
Zend_Form_Element_Password('password');
    $text->setLabel('Senha');
    $text->setAttrib('id','password');
    $text->setRequired(true);
    $text->addDecorator('HtmlTag');
    $form->addElement($text);
```

Botões de submissão

```
    $submit = new
Zend_Form_Element_Submit('incluir');
    $submit->setAttrib('value','incluir');
    $submit->setRequired(true);
    $form->addElement($submit);
```

Caixas de seleção

```
    $select = new
Zend_Form_Element_Select("porquinho");
    $select->addMultiOption('1','Cícero');
    $select->addMultiOption('2','Heitor');
    $select-
>addMultiOption('3','Prático');
    $select->setLabel('Porquinho');
    $form->addElement($select);
```

Et coetera

Claro que há mais classes, para os outros elementos HTML.

Mas tendo conhecimentos de HTML e dominando a programação orientada a objetos em PHP, fica muito simples manipular quaisquer outros componentes filhos de Zend_Form.

Validando um formulário

Os dados de um formulário gerado com Zend_Form podem ser validados com o método *isValid()*.

Um exemplo de uso é este:

```
$form = new Zend_Form();
$formValido = $form->isValid();
```

Armazenando definições de formulários

Podemos definir formulários em arquivos .ini, e carregar essas definições com a classe Zend_Config_Ini.

Isso permite uma flexibilidade muito grande na alteração de formulários, além de separar esse elemento HTML em uma outra camada, resolvendo um problema de insuficiência do padrão MVC.

Criando a definição no arquivo .ini

```
[form]
index.login.action = "index/login"
index.login.method = "post"

index.login.elements.username.type = "text"
index.login.elements.username.options.label
= "Usuário"
index.login.elements.username.id =
"username"
index.login.elements.username.options.requi
red = "true"
```

Criando a definição no arquivo .ini

```
index.login.elements.password.type =
"password"
index.login.elements.password.options.label =
"Senha"
index.login.elements.password.id = "password"
index.login.elements.password.options.required
= "true"

index.login.elements.login.type = "submit"
index.login.elements.login.value = "Login"

index.login.elements.oculto.type = "hidden"
index.login.elements.oculto.name = "cadastro"
index.login.elements.oculto.value = "perfis"
```

Carregando a definição

```
    $config = new
Zend_Config_Ini('./application/form.ini',
'form');
    $form = new Zend_Form($config->index-
>login);
    $this->_response->setBody($form);
```

Tarefa: criar formulários baseados em tabelas

Vamos criar uma classe que permita a criação de formulários baseada em parâmetros variáveis, e que construa os elementos HTML de acordo com os tipos de dados de uma tabela do banco de dados (que será o parâmetro obrigatório).

MOLEZA? Então, mãos à obra!

Tarefa: criar formulários baseados em tabelas

NÃO ENTENDI NADA!

Tarefa: criar formulários baseados em tabelas

Dicas:

- Construa uma pasta dentro do diretório **library**.
- A classe terá o nome do diretório+nome do arquivo que a contém.
- Desse modo, você pode carregá-la com Zend_Loader.
- Crie um método estático nessa classe, para não ter de instanciá-la.
- Crie um método que receba uma matriz como argumento, para poder passar parâmetros variáveis.

Referência única hoje

Documentação oficial do Zend Framework