

2009

CakePHP: Framework para Desenvolvimento WEB ágil



Thiago Rheinheimer Costa

Wesley Klewerton Guez Assunção

Tabela de Conteúdo

| | |
|---|-----------|
| Introdução ao CakePHP | 1 |
| O que é o CakePHP?..... | 1 |
| As origens do MVC..... | 1 |
| A arquitetura de uma aplicação MVC..... | 1 |
| MVC no CakePHP..... | 2 |
| Convenção de pastas do CakePHP..... | 2 |
| Iniciando a construção de uma nova aplicação | 2 |
| Views | 4 |
| Criando uma View..... | 5 |
| Prevenindo Javascript Injection..... | 5 |
| Entendendo os <i>Controllers</i> | 5 |
| Como as <i>Actions</i> são chamadas..... | 6 |
| Modelos (Models) | 7 |
| Criando um modelo de dados..... | 7 |
| Listando os registros..... | 8 |
| Retornando um registro único do banco de dados..... | 10 |
| Criando novos registros..... | 12 |
| Editando registros..... | 14 |
| Deletando registros..... | 16 |
| Validação de dados..... | 17 |
| Rotas (Routes) | 18 |
| HTML Helpers | 19 |
| O que são HTML Helpers..... | 19 |
| Usando HTML Helpers..... | 20 |

Introdução ao CakePHP

O que é o CakePHP?

O *CakePHP* é um *Framework* para desenvolvimento de aplicações web no padrão Modelos-Visões-Controladores (*Models-Views-Controllers*).

Softwares que atendem os requisitos e que são de rápida manutenção/alteração, e também aqueles que são implementados usando boas práticas de desenvolvimento e padrões, são os chamados Bons Softwares. Com o *CakePHP*, a construção de um software assim se torna fácil.

Com este *Framework*, também é possível implementar usando Test-Driven-Development (TDD), ou seja, desenvolvimento orientado a testes. Este fato permite que tudo na aplicação possa ser testado e torna essa ferramenta muito poderosa.

O uso de Test-Driven Development (desenvolvimento baseado em testes) não será abordado durante este minicurso.

As origens do MVC

Os primeiros escritos sobre esse padrão foram feitos em 1978 por Trygve Reenskaug, durante uma visita científica ao grupo *Smalltalk*. O padrão foi chamado de *Thing Model View Editor* e rapidamente teve seu nome alterado para *Model View Controller*.

A primeira implementação foi como parte da biblioteca de classes do *Smalltalk*, e era usado para criar interfaces gráficas.

A mudança radical aconteceu quando ele foi adaptado para aplicações web, e muitas vezes é chamado de *Model 2 MVC Pattern*.

Hoje o MVC também está sendo usado por aplicações web de diversas tecnologias, como o *ASP.NET MVC*, *Ruby On Rails*, *Django*, *Tapestry* (java), entre outras.

A arquitetura de uma aplicação MVC

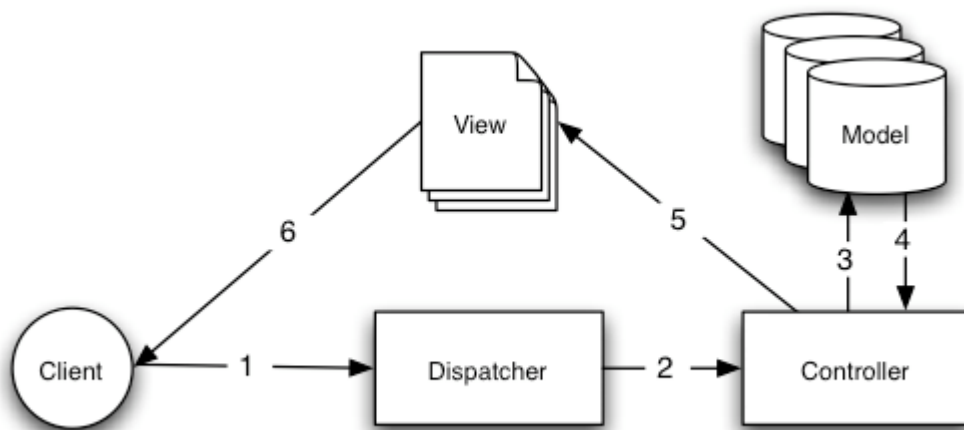
Uma aplicação baseada em MVC está dividida nas partes:

- Modelos (Models) – que contém toda a lógica da aplicação que não faz parte de uma Visão (View). Isso quer dizer que todas as regras de negócios, validações e acesso a dados estão contidas no Modelo;
- Visões (Views) – é a camada de apresentação, a interface que será mostrada para o usuário. Uma View contém todo o HTML e lógica necessários para a correta exibição das páginas aos usuários;
- Controladores (Controllers) – são responsáveis por interagir entre os modelos e as visões. Contém a lógica de controle da aplicação que não está presente nas visões e nos modelos. Controla o fluxo da aplicação.

Separar a aplicação nessas três partes é muito interessante. Você pode, por exemplo, modificar todo o layout de suas páginas sem encostar no restante da aplicação, modificando apenas as Views.

MVC no CakePHP

A arquitetura MVC no CakePHP pode ser visualizada na imagem abaixo:



Fonte: <http://book.cakephp.org/view/10/Understanding-Model-View-Controller>

Convenção de pastas do CakePHP

Quando iniciamos um novo projeto, os arquivos da nossa aplicação ficarão na pasta “app”. As principais pastas da aplicação são:

- app\config – pasta que armazena as configurações para acesso ao banco de dados, rotas, e o core da aplicação onde são configurados modo de debug entre outros;
- app\controllers – armazena os controladores da aplicação (*controllers*);
- app\models – os modelos criados na aplicação deverão ser salvos nesta pasta;
- app\views – container onde estarão todas as marcações HTML que serão renderizadas no browser do usuário. É a chamada interface do usuário;
- app\webroot – pasta pública da aplicação. É a pasta onde salvamos conteúdo público como CSS, Javascripts, Imagens, entre outros.

É extremamente importante seguir estes padrões e convenções para o correto funcionamento e organização do projeto.

Dessa maneira a aplicação se torna fácil de ser entendida, pois cada “parte” está no seu “devido lugar”.

Iniciando a construção de uma nova aplicação

O nosso primeiro passo será fazer o download da versão estável mais recente da *CakePHP*. Para isso, acesse o site <http://www.cakephp.org> e clique no botão ilustrado abaixo:



Você será encaminhado até uma página que pede que escolha entre os arquivos disponíveis. No caso de estar usando *Windows*, baixe o arquivo [cake 1.2.4.8284.zip](#) clicando sobre ele. Isso nos redirecionará para a página de doações. Se não deseja efetuar uma doação, simplesmente clique no botão “No, Thanks” que você verá então a página com um link escrito [Download Latest Release](#).

Descompacte-o e nomeie o diretório para “minicurso”, veja que você deverá estar com o diretório igual a imagem abaixo:



O conteúdo completo do diretório, incluindo os arquivos (exceto o *.eprj* que foi criado pelo *e-texteditor* usado na edição dos arquivos para este material):

| Name | Date modified |
|-----------|------------------|
| app | 16/08/2009 02:50 |
| cake | 16/08/2009 02:50 |
| vendors | 16/08/2009 02:50 |
| .eprj | 16/08/2009 02:57 |
| .htaccess | 14/06/2006 11:02 |
| index | 18/12/2008 19:16 |
| README | 25/12/2008 16:21 |

Em servidores *Apache*, basta criar um site apontando para este diretório que acabamos de criar que essa aplicação já poderá ser acessada no browser.

Para desenvolvimento no *Windows*, uma boa opção para se usar é o *Xampp for Windows* que já vem com *Apache*, *MySQL*, *PHP 5.3* e outros. Você pode baixá-lo em <http://www.apachefriends.org/en/xampp-windows.html>.

Veja parte da página que é visualizada logo após efetuar os passos anteriores:

Release Notes for CakePHP 1.2.4.8284.

[Read the changelog](#)

```
Notice (1024): Please change the value of 'Security.salt' in app/config/core.php to a salt value specific to
```

Podemos ver na imagem acima que a parte amarela traz um aviso nos dizendo que devemos trocar o *Security.salt* da nossa aplicação. Como essa configuração está setada com o valor padrão que o *Cake* traz, pois acabamos de descompactar os arquivos, devemos acessar o arquivo *app\config\core.php* e alterarmos essa configuração, veja na imagem abaixo:

```
151 | Configure::write('Security.salt', 'XYhG93b0qyJfIxfS2guVoUubWwvniR2G0FgaC9mi');
```

Esta configuração, como podemos ver na imagem, está na linha 151 do arquivo citado, e atribui um valor ao *Security.salt*. Esse hash é uma string randômica e é usado nos métodos de segurança.

Troque apenas uma das letras por outra qualquer e salve o arquivo. Recarregando a página, não veremos mais aquele aviso visto anteriormente.

Vamos aproveitar e criar o arquivo *app\views\layouts\default.ctp* com o conteúdo mostrado pela imagem abaixo:

```
1 <?php echo $html->css('cake.generic'); ?>
2 <div id="page">
3   <h2>Minicurso de CakePHP</h2>
4
5   <div id="header">
6   </div>
7
8   <div id="content">
9     <?php $session->flash(); ?>
10    <?php echo $content_for_layout; ?>
11  </div>
12
13  <div id="footer">
14  </div>
15 </div>
```

Esse será o layout padrão para todas as páginas que abrirmos na aplicação. O conteúdo das páginas que abrirmos vai aparecer no local *\$content_for_layout* exibido na figura acima.

Views

As *views* são as páginas que são renderizadas na tela do usuário.

As *Views* no *CakePHP* ficam na pasta *app\views*. Esta, por sua vez, possui uma pasta para cada controlador, para armazenar as *views* de cada um deles. Se tivermos dois controladores em nossa aplicação, *HomeController* e *EventosController*, nós teremos uma pasta em *app\views\home* que irá armazenar as *views* do respectivo controlador, e uma pasta *app\views\eventos* que armazenará as *views* do controlador *EventosController*.

Criando uma View

Para criar uma *view* basta adicionar um arquivo com extensão “.ctp” em uma pasta determinada. Se possuímos um controlador de Relatórios na aplicação, teremos provavelmente uma *view* chamada “vendas.ctp” na pasta *app\views\relatorios*. O conteúdo de uma *view* nada mais é do que marcação HTML combinada com código PHP, quando necessário.

Prevenindo Javascript Injection

Sempre que precisemos exibir uma informação na tela, devemos nos preocupar em usar o *htmlspecialchars()* do PHP para que javascript injection seja evitado.

Entendendo os *Controllers*

Um *controller* pode ser adicionado criando um novo arquivo na pasta *app\controllers* com a extensão *.php*.

O nome de um *controller* deve ser sempre seguido de *Controller*. Por exemplo, se queremos criar um controlador para fazer nossas ações de adição, exclusão e consulta de eventos nós criaremos ele com o nome *eventos_controller.php* e a classe deverá ter o nome *EventosController* pois o *Cake* tem na convenção que os nomes devem ser informados com iniciais maiúsculas e que o arquivo físico será com minúsculas, mas separado com “_” (underscore).

Em um controlador as ações não passam de métodos/funções. Cada *action* em um *controller* fica exposta para todos da internet acessem, por isso é importante termos cuidado com o que criaremos.

Veremos detalhes sobre controladores e ações mais adiante, no tópico de *Models*, pois é onde começamos a criar os *Modelos* e o restante dos nossos exemplos. Mas por agora, basta olhar a estrutura de um controlador na figura abaixo:

```

1 <?php
2
3 class EventosController extends AppController {
4     function index() { ...
7
8     function view($id = null) { ...
12
13    function add() { ...
21
22    function edit($id = null) { ...
33
34    function delete($id) { ...
39 }
40
41 ?>

```

A figura acima mostra o esqueleto de um controlador que tem como arquivo físico o `app\controllers\eventos_controller.php`. Veja que a classe é declarada como “EventosController” sem o sinal de “underscore” e deve estender a classe “AppController”.

A figura ainda nos dá a idéia de que nós faremos “ações” com os “eventos”, que são: lista (index), view (ver/detalhar), add (adicionar), edit (alterar), delete (apagar).

Como as *Actions* são chamadas

Na maioria das vezes as ações são executadas após uma requisição do usuário. Essa requisição pode ser GET ou POST ou alguma delas usando AJAX.

Nos controladores, para saber se a requisição foi feita via AJAX, devemos adicionar um componente chamado *RequestHandler*. Então basta executarmos a verificação `$this->RequestHandler->isAjax()` para termos como resposta *true* ou *false*. Veja na figura abaixo:

```

1 <?php
2
3 class EventosController extends AppController {
4     var $components = array('RequestHandler');
5
6     function testeAjax() {
7         if ($this->RequestHandler->isAjax()) {
8             ...
9         }
10    }
11

```

Da mesma maneira, podemos usar `isPost()`, `isGet()`, `isPut()`, `isDelete()` no componente *RequestHandler*.

Com esses comandos podemos controlar a execução do código dependendo do tipo da requisição.

Modelos (Models)

Podemos dizer que, basicamente, toda a lógica que não for de *views* e de *controllers* será feita nos Modelos. Nos *Models* podemos fazer validação, regra de acesso a dados e regra de negócios.

O *CakePHP* é quase que completamente baseado no modelo de dados.

Significa dizer que se criarmos nossas tabelas dando atenção às convenções de nomes que o framework usa, nos restará pouco trabalho a fazer nos Modelos, ou seja, teremos que apenas definir as associações e algumas validações.

Criando um modelo de dados

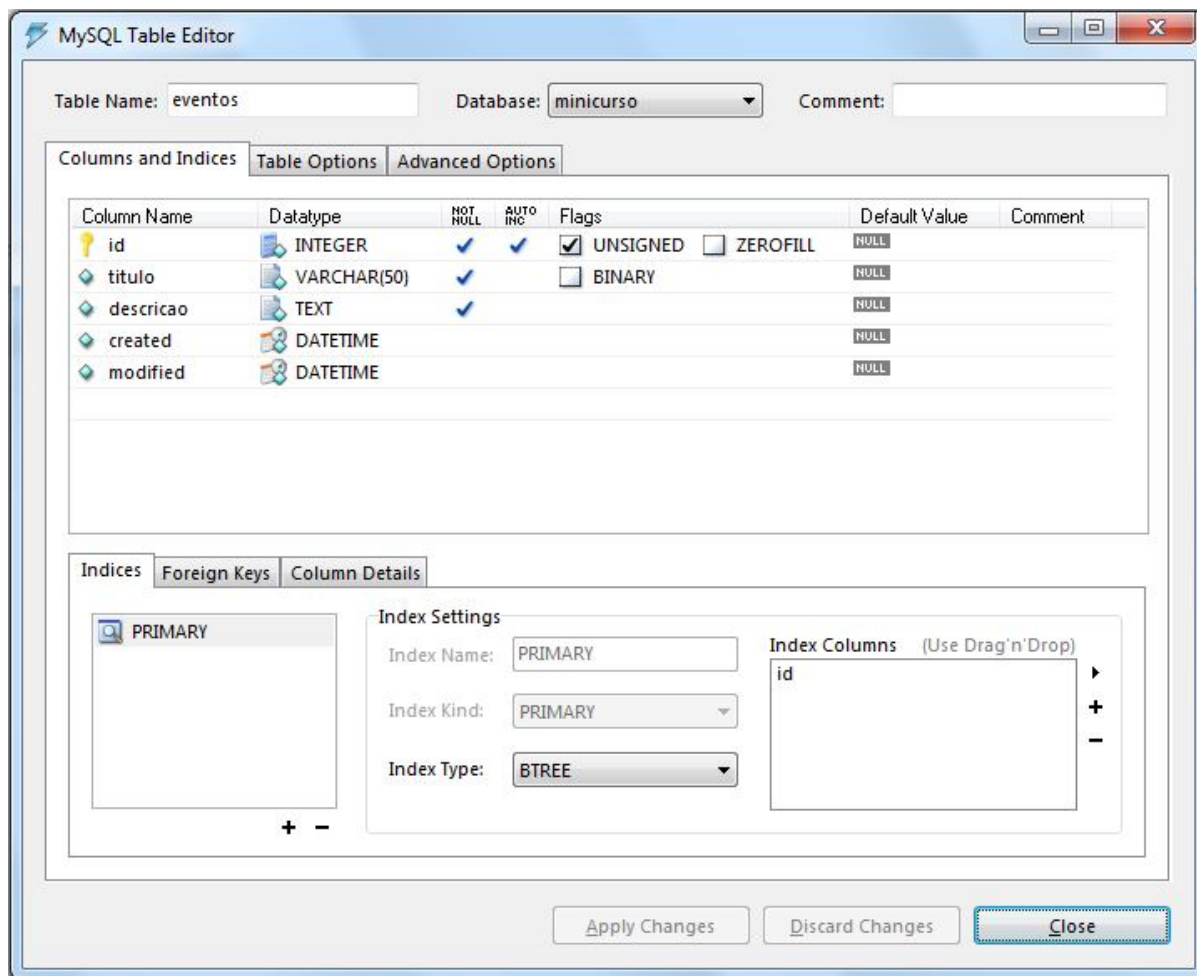
Primeiro devemos criar um banco novo no *MySQL* que será usado para o nosso minicurso. Crie um banco de dados chamado “minicurso”.

Após termos criado o banco de dados, devemos configurá-lo no arquivo *app\config\database.php*. Veja que se é a primeira vez que iremos usar este arquivo, devemos primeiramente criá-lo renomeando o arquivo *database.php.default* para *database.php*.

Veja na ilustração abaixo como nossa configuração deve estar neste momento:

```
class DATABASE_CONFIG {  
  
    var $default = array(  
        'driver' => 'mysql',  
        'persistent' => false,  
        'host' => 'localhost',  
        'login' => 'root',  
        'password' => 'senhadoroot',  
        'database' => 'minicurso',  
        'prefix' => '',  
    );  
  
    var $test = array(  
        'driver' => 'mysql',  
        'persistent' => false,  
        'host' => 'localhost',  
        'login' => 'user',  
        'password' => 'password',  
        'database' => 'test_database_name',  
        'prefix' => '',  
    );  
}
```

Agora vamos acessar o nosso banco de dados chamado “minicurso” usando o *MySQL Query Browser* e vamos criar uma tabela chamada “*eventos*”, veja a definição da tabela na ilustração abaixo:



Veja que temos os dois últimos campos chamados de “created” e “modified”. Estes campos serão gerenciados automaticamente pelo *Cake* sem termos que nos preocupar, por isso damos os nomes de acordo com as convenções usadas por ele. O campo “id” também será identificado automaticamente.

O fato de termos nomeado a tabela como “eventos” levará a termos um *Model* chamado *Evento* entre nossas classes.

Vamos então criar o arquivo `app\models\evento.php`, conforme figura abaixo:

```

1 <?php
2 class Evento extends AppModel {
3
4 }
5 ?>
```

Por termos salvo o arquivo na pasta *models* dentro da pasta *app*, e nomeado a classe como “Evento”, o *Cake* usará suas pluralizações e efetuará comandos no banco de dados supondo que a tabela seja chamada de “eventos”.

Listando os registros

Agora que temos criado um de nossos *Models*, vamos listar os registros que temos na nossa tabela de *Eventos*. (inclua alguns eventos manualmente).

A ação (action) *index()* do controle *EventosController* abaixo mostra como retornar todos os registros que temos na tabela *eventos*:

```
1 <?php
2
3 class EventosController extends AppController {
4     function index() {
5         $this->set('eventos', $this->Evento->find('all'));
6     }
7 }
8
9 ?>
```

A linha 5 é que nos interessa agora, pois ela está usando o comando “`$this->Evento->find('all')`” que faz com que todos os registros da tabela de eventos sejam retornados do banco de dados, e então eles são passados para uma variável da view chamada “eventos”.

Como a convenção de nomes fará com que o arquivo *app\views\eventos\index.ctp* seja renderizado no browser do usuário, basta criar esse arquivo que terá a lógica de exibição desses dados, ou seja, um famoso “foreach” do PHP que mostra os dados da variável “\$eventos”.

Veja o código do arquivo *app\views\eventos\index.ctp* na imagem abaixo:

```
1 <h3>View index() do Controller "EventosController"</h3>
2
3 <?php pr($eventos); ?>
```

Ao abrirmos, no browser, o endereço dessa nossa aplicação, <http://localhost/eventos>, veremos algo semelhante à figura abaixo:

Minicurso de CakePHP

View index() do Controller "EventosController"

```
Array
(
    [0] => Array
        (
            [Evento] => Array
                (
                    [id] => 1
                    [titulo] => Minicurso de CakePHP
                    [descricao] => Muito interessante!
                    [created] => 2009-08-16 04:38:53
                    [modified] =>
                )
            )
        [1] => Array
            (
                [Evento] => Array
                    (
                        [id] => 2
                        [titulo] => Minicurso de XXX
                        [descricao] => Legal!!
                        [created] => 2009-08-16 04:39:11
                        [modified] =>
                    )
            )
)
```

Retornando um registro único do banco de dados

Quando criamos actions para ver detalhes de um evento, por exemplo, ou até mesmo ao editar um registro, nós devemos retornar um único registro do banco de dados, e não uma coleção deles. Nesse caso nós podemos usar o código abaixo:

```

1 <?php
2
3 class EventosController extends AppController {
4     function index() {
5         $this->set('eventos', $this->Evento->find('all'));
6     }
7
8     function detalhe($id = null) {
9         $this->Evento->id = $id;
10        $this->set('evento', $this->Evento->read());
11    }
12 }
13
14 ?>

```

Veja que a função “detalhe” está circulado em vermelho. Quando o usuário acessar <http://localhost/eventos/detalhe/1> ele verá o conteúdo da *view* “detalhe.ctp” que deve estar localizada na pasta *app\views\eventos\detalhe.ctp*, conforme figura abaixo:

```

1 <h3>View detalhe() do controller "EventosController"</h3>
2
3 <?php pr($evento); ?>

```

Isso mostrará apenas o registro que o usuário esteja solicitando. Veja como a página de detalhe de nosso exemplo ficou:

Minicurso de CakePHP

View detalhe() do controller "EventosController"

```

Array
(
    [Evento] => Array
        (
            [id] => 1
            [titulo] => Minicurso de CakePHP
            [descricao] => Muito interessante!
            [created] => 2009-08-16 04:38:53
            [modified] =>
        )
)

```

Veja que apenas o evento 1 está sendo exibido, ao contrário da *view* “index()” que exibe todos os eventos que tiverem no banco de dados.

Criando novos registros

Para criar novos registros nós devemos primeiro mostrar um formulário ao usuário. Só após o usuário efetuar o POST desse formulário é que ele deverá ser salvo no banco de dados.

O código do nosso “EventosController”, ilustrado na figura abaixo, mostra uma nova ação (function/action) chamada “adicionar” que verifica se há dados postados pelo usuário e, se houver, tenta salvá-los no banco de dados e redireciona o usuário para a ação “index()”, do contrário apenas mostrará a *view* para o usuário preencher o formulário, considerando que ele queira adicionar um novo evento:

```
1 <?php
2
3 class EventosController extends AppController {
4     function index() {
5         $this->set('eventos', $this->Evento->find('all'));
6     }
7
8     function detalhe($id = null) {
9         $this->Evento->id = $id;
10        $this->set('evento', $this->Evento->read());
11    }
12
13    function adicionar() {
14        if (!empty($this->data)) {
15            if ($this->Evento->save($this->data)) {
16                $this->Session->setFlash('O evento foi adicionado!');
17                $this->redirect(array('action' => 'index'));
18            }
19        }
20    }
21 }
22
23 ?>
```

O código também define uma mensagem para ser exibida ao usuário em caso de sucesso, dizendo: “O evento foi adicionado!”, e, logo após, redireciona-o para a ação “index()” que no nosso caso nos mostrará novamente a lista de todos os eventos.

Vamos agora criar o formulário para a inclusão do evento. Crie o arquivo `app\views\eventos\adicionar.ctp` com o código conforme o da figura abaixo:

```

1 <?php echo $form->create('Evento');?>
2 <fieldset>
3 <legend><?php __('Informe os dados do evento e clique em Adicionar');?></legend>
4 <?php
5 echo $form->input('titulo', array('label' => 'Título'));
6 echo $form->input('descricao', array('label' => 'Descrição'));
7 ?>
8 </fieldset>
9 <?php echo $form->end('Adicionar');?>
10

```

Após salvar o arquivo acima podemos abrir nossa aplicação em <http://localhost/eventos/adicionar> e veremos uma página semelhante à ilustrada pela figura abaixo:

Minicurso de CakePHP

Informe os dados do evento e clique em Adicionar

Título

Descrição

Adicionar

Veja que o *Cake* detectou que o campo descrição é um “textarea” automaticamente, pois no banco de dados criamos ele como “TEXT”.

Preencha um título e uma descrição e clique no botão “Adicionar”.

Veja que isso causou um erro.

De acordo com as convenções de nome do *Cake*, ao criar um form para um determinado *Model* ele supõe que a *action* para adicionar é *add*, apagar é *delete* e para editar é *edit* e, finalmente, para visualizar/detalhar é *view*. No nosso caso, criamos a ação com o nome

adicionar e o *Framework* está dando o POST para a ação *add* que não existe no nosso “EventosController”.

Para corrigir este problema podemos alterar nossa ação e nossa view para “add” ao invés de “adicionar”. Outra maneira é alterar o código do arquivo *app\views\eventos\adicionar.ctp* para o código mostrado na figura abaixo:

```
1 <?php echo $form->create('Evento', array('action' => 'adicionar'));?>
2 <fieldset>
3     <legend><?php __('Informe os dados do evento e clique em Adicionar');?></legend>
4     <?php
5         echo $form->input('titulo', array('label' => 'Título'));
6         echo $form->input('descricao', array('label' => 'Descrição'));
7     ?>
8 </fieldset>
9 <?php echo $form->end('Adicionar');?>
10
```

Veja que na primeira linha, onde criamos o formulário, devemos informar qual a ação que é responsável por receber os dados preenchidos.

Agora basta abrir novamente o <http://localhost/eventos/adicionar>, preencher o formulário e clicar em adicionar. Após feito isso, seremos redirecionados para */eventos* que nos mostrará a lista dos eventos incluindo o que acabamos de lançar!

Edi tando regi stros

Vamos adicionar uma *action* chamada *edit* ao nosso *EventosController*, veja o código na figura abaixo:

```

1 <?php
2
3 class EventosController extends AppController {
4     function index() {
5         $this->set('eventos', $this->Evento->find('all'));
6     }
7
8     function detalhe($id = null) {
9         $this->Evento->id = $id;
10        $this->set('evento', $this->Evento->read());
11    }
12
13    function adicionar() {
14        if (!empty($this->data)) {
15            if ($this->Evento->save($this->data)) {
16                $this->Session->setFlash('0 evento foi adicionado!');
17                $this->redirect(array('action' => 'index'));
18            }
19        }
20    }
21
22    function edit($id = null) {
23        $this->Evento->id = $id;
24        if (empty($this->data)) {
25            $this->data = $this->Evento->read();
26        } else {
27            if ($this->Evento->save($this->data)) {
28                $this->Session->setFlash('0 evento foi alterado!');
29                $this->redirect(array('action' => 'index'));
30            }
31        }
32    }
33 }
34
35 ?>

```

Agora basta criarmos o arquivo `app\views\eventos\edit.ctp` com o código igual o da imagem abaixo:

```

1 <?php echo $form->create('Evento');?>
2     <fieldset>
3         <legend><?php __('Digite as alterações abaixo e clique em Salvar');?></legend>
4         <?php
5             echo $form->input('id');
6             echo $form->input('titulo', array('label' => 'Título'));
7             echo $form->input('descricao', array('label' => 'Descrição'));
8         ?>
9     </fieldset>
10 <?php echo $form->end('Salvar');?>

```

Veja que neste caso, na primeira linha, diferentemente do que fizemos no arquivo `adicionar.ctp`, nós não precisamos informar para qual `action` o formulário será enviado, pois criamos ela de acordo com as convenções adotadas pelo *Cake*. Outro destaque é o da linha 5,

como estamos alterando um registro devemos colocar o *id* no formulário para que o *Framework* saiba montar a SQL de alteração.

Agora podemos acessar <http://localhost/eventos/edit/1> e fazermos uma alteração e clicarmos em Salvar.

Deletando registros

O correto, na deleção de registros, é exibir uma página para o usuário pedindo a confirmação se ele deseja ou não remover e o botão de confirmação ser um POST para a ação. Isso evita que buscadores que acessem o link via GET deletem seus registros.

Vamos criar uma ação chamada *delete* no nosso *EventosController*, veja na figura abaixo:

```
1 <?php
2
3 class EventosController extends AppController {
4     function index() {
5         $this->set('eventos', $this->Evento->find('all'));
6     }
7
8     function detalhe($id = null) {
9         $this->Evento->id = $id;
10        $this->set('evento', $this->Evento->read());
11    }
12
13    function adicionar() {
14        if (!empty($this->data)) {
15            if ($this->Evento->save($this->data)) {
16                $this->Session->setFlash('0 evento foi adicionado!');
17                $this->redirect(array('action' => 'index'));
18            }
19        }
20    }
21
22    function edit($id = null) {
23        $this->Evento->id = $id;
24        if (empty($this->data)) {
25            $this->data = $this->Evento->read();
26        } else {
27            if ($this->Evento->save($this->data)) {
28                $this->Session->setFlash('0 evento foi alterado!');
29                $this->redirect(array('action' => 'index'));
30            }
31        }
32    }
33
34    function delete($id) {
35        $this->Evento->delete($id);
36        $this->Session->setFlash('0 evento foi removido!');
37        $this->redirect(array('action' => 'index'));
38    }
39 }
40
41 ?>
```

Isso nos permite acessar o endereço <http://localhost/eventos/delete/1> que removerá o evento com *id=1* do banco de dados e nos redirecionará para a lista de eventos mostrando uma mensagem que “O evento foi removido!”.

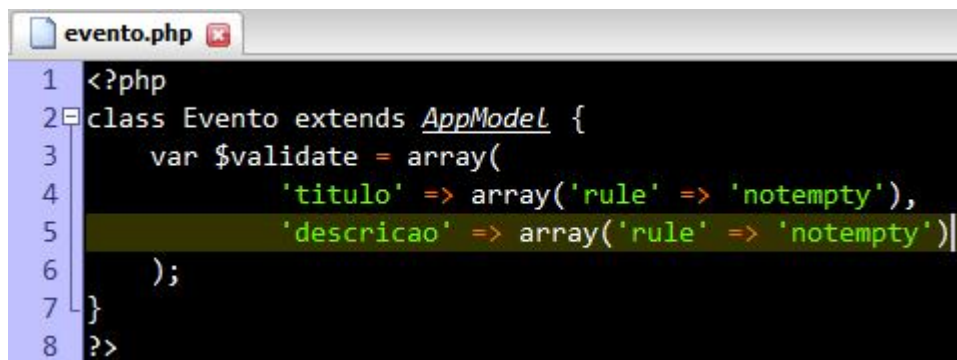
Veja que no exemplo não usamos uma verificação do tipo da requisição para deletar. Como explicado anteriormente, o correto seria verificar com (`!$this->RequestHandler->isGet()`) para ter certeza de que a requisição não foi via GET e caso tenha sido, exibir uma página de confirmação.

Validação de dados

Sim, já podemos adicionar, apagar, alterar e listar nossos eventos. Mas e o que acontece se tentarmos salvar um evento sem preencher o título e a descrição? Se fizermos o teste veremos que um novo evento é incluído com o título e a descrição em branco. Isso não é um bom sinal, é? Está faltando alguma coisa?

Sim! Está faltando validação de dados!

De acordo com o MVC, a validação dos dados não deve ser feita nas visões e nem nos controladores. Os responsáveis pelas regras de negócios e validações são os *Models*, então vamos dizer aos nossos models algumas restrições que devem ser verificadas antes de salvar um evento! Veja o código ilustrado na figura abaixo:



```
evento.php
1 <?php
2 class Evento extends AppModel {
3     var $validate = array(
4         'titulo' => array('rule' => 'notEmpty'),
5         'descricao' => array('rule' => 'notEmpty')
6     );
7 }
8 ?>
```

O arquivo acima é o `app\models\evento.php`. Adicionamos validação para o título e a descrição, dizendo que não podem ser vazios.

Após tentar novamente incluir um evento sem informar o título, vemos algo semelhante ao que segue abaixo:

Minicurso de CakePHP

Informe os dados do evento e clique em Adicionar

Título

This field cannot be left blank

O problema aqui é que a mensagem é automática e em inglês. Para informarmos nossa própria mensagem, podemos modificar um pouco a validação, como segue na figura abaixo:

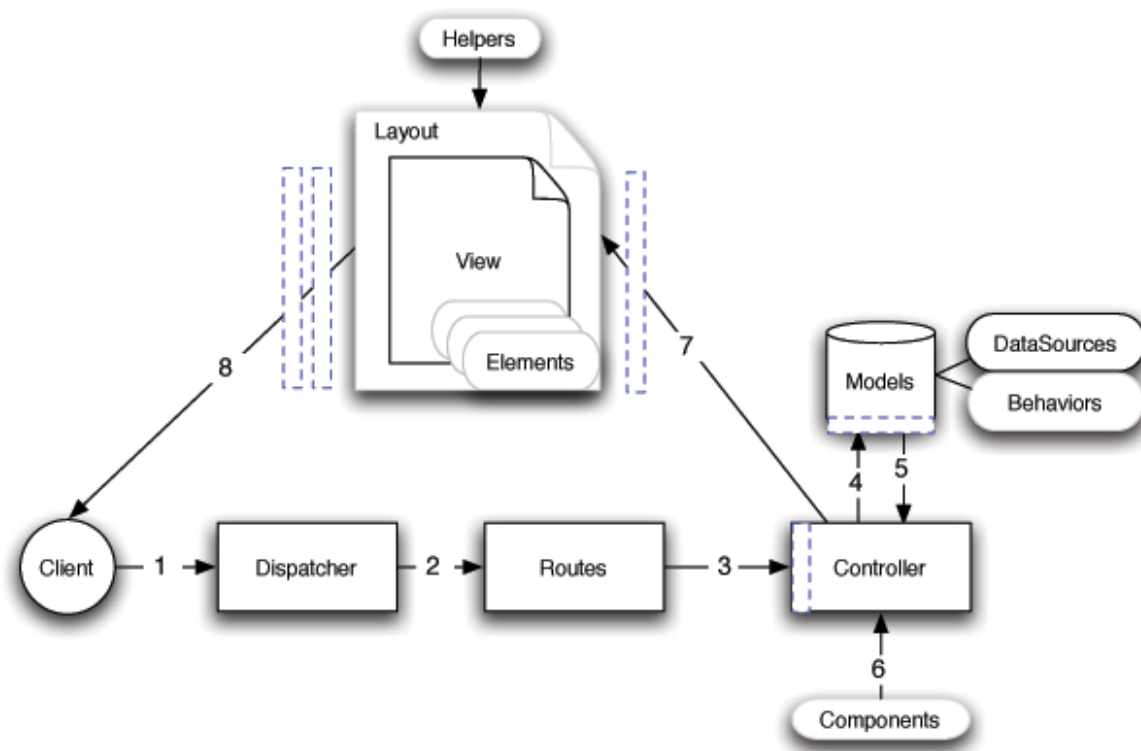
```
evento.php
1 <?php
2 class Evento extends AppModel {
3     var $validate = array(
4         'titulo' => array('rule' => 'notEmpty', 'message' => 'O título é obrigatório!'),
5         'descricao' => array('rule' => 'notEmpty')
6     );
7 }
8 ?>
```

Há inúmeras validações que podem ser feitas além dessa. Pode ser validado usando regular expression, condições sql, tamanho do campo, entre outros.

Para detalhamento completo de todas as validações disponíveis acesse <http://book.cakephp.org>.

Rotas (Routes)

As rotas são os mapeamentos de requisições feitas pelo Browser para um controlador e uma ação específicos. No Cake, a requisição feita pelo usuário é enviada ao dispatcher que localiza a rota que será usada e passa essa requisição para o controlador, executando uma determinada ação deste último. Veja a ilustração abaixo:



Fonte: <http://book.cakephp.org/view/21/A-Typical-CakePHP-Request>

O arquivo para a configuração das rotas, no Cake, é o arquivo *routes.php* que se encontra no diretório *app\config\routes.php*.

Então é através das rotas que nossa aplicação sabe qual ação de qual controlador é para ser executada quando determinada requisição é efetuada. Se um usuário acessar <http://www.meusite.com> é muito provável que a página que ele visualize seja uma view chamada “index” em um controller chamado “home”. Para isso, nosso arquivo de rotas deve ter uma configuração parecida com a abaixo:

```
Router::connect('/', array('controller' => 'home', 'action' => 'index'));
```

Por padrão, no CakePHP, a ação *index* é chamada em um *controller* quando nenhuma ação é passada na requisição. Por isso se acessamos */eventos* da nossa aplicação, será executada a ação *index* do *EventosController*.

HTML Helpers

O que são HTML Helpers

HTML Helpers podem tornar a tarefa de fazer views mais fácil. Eles simplesmente renderizam algum código HTML que pode ser usado em todas as suas *views* simplesmente chamando `<?php echo $html->nome_do_helper(parametros); ?>`. Alguns *Helpers* já estão incluídos no *CakePHP*, e outros podem ser customizados de acordo com as nossas necessidades.

Usando HTML Helpers

Há inúmeros helpers que já estão prontos para serem usados. Não só HTML Helpers como também outros helpers que renderizam forms, trabalham com cache, com requisições ajax, com paginação, RSS, Session, Javascript, entre outros.

Ao usar, por exemplo, `$html->link('Texto do link', array('controller' => 'eventos', 'action' => 'index'))`, nós estaremos fazendo um atalho para um `...`.

Para uma lista completa de todas as possibilidades, podemos acessar o site <http://book.cakephp.org>.