

Guia do Desenvolvedor

# Bem vindo

Seja bem vindo ao **Guia do Desenvolvedor EasyFramework**! Através desse guia, você aprenderá como o EasyFramework pode ajudar a melhorar a forma como você desenvolve suas aplicações, esquecendo da parte repetitiva e tediosa e mantendo apenas a parte criativa do seu trabalho. Além disso, se você já programa há algum tempo usando EasyFramework, esse guia também servirá para que você conheça outras funcionalidades interessantes adicionadas nas últimas versões do framework ou aprofunde o conhecimento no que você já conhece.

## Começando

Se você é novo por aqui, talvez você queira descobrir um pouco mais sobre [o que o EasyFramework pode fazer por você](http://spaghettiphp.com.br/docs/developer-guide/por-que), e por que utilizá-lo fará seu trabalho mais produtivo e divertido. Se você já tem alguma experiência com o EasyFramework, você pode [descobrir o que há de novo](http://spaghettiphp.com.br/docs/developer-guide/o-que-ha-de-novo) desde a última versão, ou navegar direto para o assunto que você está procurando através da barra ao lado.

## Requisitos

Esse guia assume que você tem uma base de conhecimentos em PHP e um entendimento básico de programação orientada a objetos. Conhecimentos sobre a arquitetura MVC ou algum framework PHP serão de alguma ajuda, mas não são requeridos para que você aprenda a usar o EasyFramework.

O EasyFramework é uma ferramenta criada para quem já sabe programar. Apesar de ser extremamente simples, os conhecimentos da linguagem são essenciais no dia-a-dia do desenvolvimento, requerendo alguma experiência prévia. Por essa razão, não é recomendável o uso do EasyFramework para pessoas que entraram há pouco tempo no mundo da programação web, nem para novatos no PHP. Nesse caso, recomendamos antes acumular alguma experiência com a linguagem.

# Por que usar o EasyFramework?

EasyFramework é fácil de aprender de utilizar. O framework esconde todo o código complexo e monótono, deixando que você se foque apenas na parte importante do desenvolvimento de suas aplicações. O código gerado no uso do EasyFramework é simples de compreender, e você precisará codificar só o necessário, evitando erros e aumentando sua produtividade.

Para que você use o EasyFramework, você não precisa aprender novas linguagens de template nem entender complexos arquivos de configuração em XML. Tudo o que você precisa é o seu conhecimento em PHP e Programação Orientada a Objetos. As views do EasyFramework são escritas usando a abstração do Smarty e HTML que você já conhece, e as poucas configurações necessárias para o framework são feitas com a própria linguagem.

Falando em configurações, as configurações no EasyFramework foram trazidas para o mínimo, e preferimos convenções em vez de configurações. Basta instalar o EasyFramework, configurar a conexão com o banco de dados e sua aplicação já estará funcionando. Entretanto, isso não tira a capacidade de customizar várias funcionalidades do framework, que é facilmente extensível em vários aspectos.

O EasyFramework foi todo escrito visando conformidade com o PHP 5. Isso significa que você terá um framework que não possui código suportando versões legadas do PHP, e tira todo o proveito das funcionalidades do PHP 5.2 e 5.3.

O EasyFramework também possui uma documentação compreensiva de todas as funcionalidades disponíveis. Com isso, você não precisa entender o código interno do framework para poder trabalhar, e sempre que você tiver problemas terá onde procurar ajuda.

Para os que pretendem mergulhar no código do EasyFramework, nosso código é limpo, conciso, bem formatado e bem documentado, permitindo que outros desenvolvedores possam fazer alterações no código sem problemas, seja para adicionar novas funcionalidades no framewrok, corrigir bugs ou personalizar o código para seu próprio uso. O EasyFramework é disponibilizado sob a [MIT License](http://www.opensource.org/licenses/mit-license.php), permitindo que você altere o código e também o redistribua segundo os termos da licença.

# Entendendo o MVC

O EasyFramework utiliza-se do design pattern MVC, separando as camadas da sua aplicação, tornando o desenvolvimento e manutenção muito mais fáceis e rápidos. Toda a camada de lógica e dados fica separada da camada de apresentação de sua aplicação, mantendo a lógica em PHP longe do código HTML. Na prática, isso acaba com a guerra entre desenvolvedores e designers, que agora podem trabalhar simultaneamente de maneira harmoniosa e eficiente, sem quebrarem o trabalho um do outro.



## Modelos de Dados (Model)

Os modelos de dados, ou Models, representam os dados de sua aplicação. Ou seja, é através do Model que sua aplicação se conectará com o banco de dados, fazendo consultas e retornando e inserindo dados.

As classes do EasyFramework irão auxiliá-lo na manipulação desses dados, sem que você precise escrever uma única linha de SQL. Todas as condições e dados a serem enviados ao banco são montados através de arrays simples, e o framework fará o trabalho pesado enquanto você se preocupa com o que realmente interessa.

Embora o EasyFramework só venha com um datasource para MySQL no momento, é possível que você crie datasources para qualquer banco de dados existente, além de poder criar datasources para outros tipos de fontes de dados, como documentos XML, por exemplo.

## Controladores (Controller)

É nos controladores, ou Controllers, que a mágica da sua aplicação realmente acontece. É aqui onde você irá tratar as respostas do usuário, buscar e processar dados e retorná-los para as Views. Nada de código HTML, nada de CSS, nada de JavaScript. Única e puramente código PHP, sem misturar as coisas.

Dentro dos controllers, as funcionalidades são separadas em métodos, que são chamados actions. Cada action corresponde a uma ação dentro de sua aplicação, e geralmente cada ação possui uma [view](http://spaghettiphp.com.br/docs/developer-guide/mvc" \l "mvc/views) relacionada. Cada action é responsável por uma pequena funcionalidade de sua aplicação, fazendo com que ela seja modular, e cada ação não se misture com as outras.

Mesmo você estando no comando dos controllers, o EasyFramework irá lhe auxiliar dando acesso aos seus Models e facilitando o envio de dados para as Views. Para isso nosso framework trabalha com a camada de abstrção do Smarty, facilitando muito o desenvolvimento das views. Você se diverte programando, e o EasyFramework o ajuda com o resto. Além disso, já existem vários componentes que o ajudarão a resolver problemas comuns no desenvolvimento, como upload de arquivos, redimensionamento de imagens, entre outros. Caso você encontre uma tarefa comum que ainda não possua um componente, o EasyFramework lhe fornece uma forma extremamente simples para a criação de novos componentes.

## Visualizações (View)

As Views fazem parte da camada de apresentação de sua aplicação. Nessa camada você se preocupa unicamente em mostrar os dados ao usuário, usando apenas a linguagem de marcação do Smarty, evitando aquela típica mistura de lógica e apresentação em um mesmo arquivo.

Além disso, também é possível usar helpers que ajudam na criação de formulários, links e inserção de imagens, formatação de texto, entre outros, sem que você precise se preocupar com detalhes chatos de implementação. Caso não exista um helper já existente para sua necessidade específica, o EasyFramework também dispõe de uma arquitetura simples para sua criação.

Escolhemos o MVC por ser um pattern que faz bastante sentido no desenvolvimento Web, além de aumentar a produtividade, diminuir a repetição de código, facilitar a correção de bugs e tornar a vida dos desenvolvedores um pouco menos conflitante, especialmente quando se trabalha em conjunto com designers. E você, está pronto para entrar brincadeira, e se unir a nós por um desenvolvimento melhor?

# Instalação

O EasyFramework é rápido e fácil de instalar. Os requerimentos mínimos são um servidor web e uma cópia do EasyFramework, comente isso!

A instalação consiste nos seguintes passos:

* Realizar o download de uma cópia do EasyFramework
* Configurar seus servidor web para lidar com o PHP, se necessário

**Fazendo o Download do EasyFramework**

Existem duas maneiras de baixar o EasyFramework. Você pode baixar diretamente pelo site um pacote zipado (zip/tar.gz/tar.bz2), ou checar nosso repositório no Sourceforge.

Para baixar o EasyFramework, visite nosso site e clique em “Baixar Agora”. Visite o site <http://www.easyframework.org>.

## Estrutura de Pastas

A estrutura de pastas do EasyFramework foi feita para simplificar a alocação de arquivos dentro da aplicação, seguindo a seguinte estrutura:

* /NomeDaApp
  + /app
    - /config
    - /controllers
    - /models
    - /views
    - /webroot
  + /core
  + /.htaccess
  + /index.php

A partir desse ponto o framework está pronto para receber sua aplicação, de forma simples e rápida.

# Criando uma App

Ao contrário do que você possa imaginar, a configuração de um framework não precisa ser longa e tediosa, tomando vários minutos a cada nova aplicação. O EasyFramework foi desenvolvido preferindo convenção ao invés de configuração, e isso significa que o ele já adota vários padrões, sem que você precise definir onde fica cada coisa e como o framework deve funcionar.

Se você pretende criar uma aplicação que não utilize banco de dados, o EasyFramework está pronto para rodar assim que ele for instalado. Como sabemos que isso raramente acontece, a única configuração básica para que uma aplicação comum funcione é a configuração de banco de dados.

## Configurando o banco de dados

Para configurar seu banco de dados, você precisa editar o arquivo /app/config/database.php. O EasyFramework já vem com algumas configurações de exemplo como mostrado abaixo, então você só precisará alterar os campos necessários.

Config::write ("database", array (

"development" => array(

"driver" => "mysql",

"host" => "",

"user" => "",

"password" => "",

"database" => "",

"prefix" => ""

),

"production" => array(

"driver" => "",

"host" => "",

"user" => "",

"password" => "",

"database" => "",

"prefix" => ""

)

));

A configuração de banco de dados consiste em vários environments, ou ambientes de desenvolvimento, cada um contendo configurações específicas para os banco de dados. Os environments existem para que você não precise mudar as configurações de seu banco de dados a cada deploy da aplicação. O EasyFramework já traz como exemplo os ambientes development, para quando a aplicação está em desenvolvimento, e production, para a aplicação em produção. Você pode criar quantos environments forem necessários, não sendo necessário limitar-se aos já pré-definidos.

Em cada configuração, você pode definir as configurações necessárias para o banco de dados. São elas:

* driver define o datasource que o EasyFramework deverá usar para se conectar com o banco de dados. mysql atualmente é o único disponível com a instalação padrão.
* host define o endereço no qual se encontra o servidor de banco de dados.
* user define o nome de usuário usado para a conexão.
* password define a senha do usuário usado para a conexão.
* database define o nome do banco de dados em que o EasyFramework irá se conectar.
* prefix define o prefixo que as tabelas do banco de dados estão utilizando. Normalmente esse prefixo é usado apenas para evitar colisão com tabelas pré-existentes no banco de dados.

## Escolhendo o Ambiente de Desenvolvimento

Em /app/config/settings.php, você pode escolher o environment a ser usado com o EasyFramework através da configuração environment:

|  |  |
| --- | --- |
| Config::write("environment", "development"); |  |

Você também pode usar essa configuração dentro de sua aplicação, assim como qualquer outra que você deseje escrever.

## E o que mais?

Você realmente esperava mais? Sentimos em desapontá-lo, mas é só isso mesmo. Há algumas outras configurações que também podem ser usados, e eles estão documentados no próprio arquivo de configuração. Eles também são citados com mais detalhes ao longo do manual.

Agora sim, estamos prontos pra começar a colocar a mão na massa!

# As classes Models

Os models do EasyFramework são os responsáveis por todas as tarefas de banco de dados. É através deles que você criará, atualizará, apagará e buscará registros das tabelas de seu banco de dados. Tudo isso sem escrever uma única linha de SQL, já que o Spaghetti\* abstrai tudo isso em um formato simples de arrays, puramente em PHP, como você verá em breve.

## Criando Models

Dentro da estrutura de diretórios do EasyFramework, os models se encontram em /app/models/. É lá onde ficarão todos os modelos de sua aplicação que você criar. O nome de arquivo de um modelo deve seguir a convenção nome\_do\_modelo.php. Por exemplo, se você criar um modelo UserPermissions, por exemplo, o nome do respectivo arquivo deverá ser user\_permissions.php.

Inicialmente, um modelo é apenas uma classe vazia, que estende de Model:

class Users extends Model {

}

Quando seu modelo herda de Model, você terá ao seu dispor toda as funcionalidades necessárias para lidar com o banco de dados.

Caso você tenha um modelo que tenha sua respectiva tabela no banco de dados com um nome diferente, ou mesmo que não possua uma tabela, é possível sobrescrever o comportamento do EasyFramework através do table.

class Users extends Model {

function \_\_construct() {

public $table = 'users\_table';

}

}

Caso nosso modelo fosse usado apenas para validação de dados, e não possuísse uma tabela no banco de dados, é só não definirmos nenhuma tabela para o atributo table;

## Apagando Registros

Model::delete() apaga um ou mais registros do banco de dados a partir do valor de sua chave primária, passada por primeiro parâmetro. Por exemplo, para apagarmos um usuário de id = 3, fazemos a seguinte chamada:

$this->model->delete(3);

Caso a operação tenha sido executada com sucesso, o método retorna true, e false caso algo de errado tenha ocorrido.

Por padrão, Model::delete() apagará todos os [registros relacionados](http://spaghettiphp.com.br/docs/developer-guide/models/relacionamentos) ao registro sendo apagado. Para evitar que isso ocorra, e que os registros dependentes sejam mantidos, passe o segundo parâmetro com o valor false.

O EasyFramework também permite que você apague vários registros de uma só vez, a partir de determinada condição. Para isso, usamos o as filtragens, recebendo como parâmetro as opções para a exclusão. Por exemplo, o código abaixo apagará os primeiros 50 registros com data anterior a 01/12/2009:

$this->model->delete (array (

“condition”=> “data < 2009-12-01 LIMIT 50”

));

torna a manutenção e compreensão de suas aplicações muito mais simples, e o código passa a dizer mais e a fazer mais sentido

## Contando Registros

Para a contagem de registros, o EasyFramework usa o método Model::count(). O método recebe como parâmetro um array de opções, indicando as condições da contagem e os campos a serem contados.

$this->model->count();

A definição dos campos usados na contagem é útil quando precisamos contar registros distintos, por exemplo:

$this->model->count (array (“fields”=>”DISTINCT id”));

Também podemos definir uma filtragem para contagem simplismente adicionando aoarray o seguinte código:

$this->model->count (array (“condition”=>”NOT NULL”));

## Fazendo consultas diretamente ao banco de dados

Você pode ralizar consultas diretamente pelo seu modelo, facilmente, apenas utilizando a função Model::query(), passando como parâmetro a consulta SQL desejada. Essa função ira retornar o resultado da consulta ou falso em caso de erro.

Exemplo:

$this->query(“SELECT \* FROM tabela”);

## Fazendo consultas e inicializando objetos

É possível realizar consultas que inicalizam objetos interios. Para isso basta que sua classe sobreescreva o método Model::inicialize(), nesse método você deverá inicializar seu objeto de acordo com a consulta passada no método Model::find\_by\_sql (), automaticamente qunado essa função é chamada, após a consulta o método inicialize é chamado passando o resultado como parâmtro, e retornando para você um objeto inicializado.

Exemplo:

$usuario = $this-> find\_by\_sql (“SELECT \* FROM users”);

function inicialize($record){

return new User(isset($record['id'])?$record['id']:null,

isset($record['username'])?$record['username']:null,

isset($record['pass'])?$record['pass']:null

);

}

**Transactions**

Transactions mantém a integridade dos dados em suas tabelas, evitando dados inconsistentes em caso de algum erro durante a criação ou alteração de registros. O EasyFramework usa automaticamente o *autocommit()* para você não precisar se preocupar com as transações do banco. É possível você fazer as transações manualmente.

# As classes Controllers

No EasyFramework , os controllers receberão as requisições do usuário, farão algum processamento, farão contato com [models](http://spaghettiphp.com.br/docs/developer-guide/models/index), enviando os dados do usuário ou recebendo dados a serem mostrados em [views](http://spaghettiphp.com.br/docs/developer-guide/views/index). São os controllers os responsáveis por juntar as partes de lógica de negócio e a visualização de sua aplicação.

## Criando Controllers

Para criarmos um controller, é preciso definir a classe <nome>Controller, e salvá-la em /app/controllers/<nome>\_controller.php. Também é necessário estender a classe AppController, para que todos os métodos necessários ao seu controller sejam herdados das classes básicas do EasyFramework, e assim sua aplicação funcionará corretamente.

|  |  |
| --- | --- |
| 1 | class PostsController extends AppController { } |

A partir de agora, a URL /posts já está disponível a partir da raíz de sua aplicação, embora o Spaghetti\* ainda retorne um erro dizendo que uma action ainda não existe. Precisamos resolver isso!

## Criando Actions

Actions nada mais são do que métodos dentro de uma classe de controller. Cada action corresponde a uma ação diferente de sua aplicação e, consequentemente, uma nova URL. Ações podem ser o envio de um e-mail, o cadastro de um usuário, a postagem de um artigo, a visualização de um perfil, além de qualquer outra ação necessária.

Para que sua aplicação funcione, é necessária a criação de actions e suas respectivas views. Vamos criar uma action simples, para gerar uma lista de posts. Essa action estará acessível através de /posts/index, ou apenas /posts, já que o Spaghetti\* subentende que, quando uma action não é definida, ela é considerada como index.

|  |  |  |
| --- | --- | --- |
| 1 | class PostsController extends AppController { | |
| 2 | public function index() { |

|  |  |  |
| --- | --- | --- |
| 3 | $this->set("posts", $this->Posts->all()); | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |

A primeira coisa que devemos saber para podermos interagir entre controllers e views é como se faz a passagem de variáveis. Como variáveis locais perderiam-se no caminho, usa-se o método Controller::set(), passando como parâmetros o nome da variável e seu respectivo valor. Dentro da view, essa variável estará disponível como uma variável local comum, acessível por $posts, no nosso caso. Caso você precise passar várias variáveis para a view, Controller::set() suporta a notação de arrays:

|  |  |  |
| --- | --- | --- |
| 1 | class PostsController extends AppController { | |
| 2 | public function index() { |

|  |  |
| --- | --- |
| 3 | $this->set(array( |
| 4 | "posts" => $this->Posts->all(), | |

|  |  |  |
| --- | --- | --- |
| 5 | "search" => $this->param("search") | |
| 6 | )); |

|  |  |  |
| --- | --- | --- |
| 7 | } | |
| 8 | } |

Por padrão, o Spaghetti\* carrega um model com o mesmo nome do controller sendo criado. Como você pode notar, o modelo Posts já está disponível em $this->Posts (isto é, se ele tiver sido criado em /app/models/posts.php). Através dele, você tem acesso a todas as [funcionalidades dos models](http://spaghettiphp.com.br/docs/developer-guide/models/index).

A partir de agora, se uma view já tiver sido criada em /app/views/posts/index.htm.php, você já poderá ver o resultado em /posts. Como dito acima, se nenhuma action é passada pela URL, o Spaghetti\* supõe que esta seja index. Qualquer outra action pode ser acessada através da estrutura /controller/action.

## Recebendo parâmetros

Assim como qualquer outra página PHP comum, uma action também pode receber parâmetros através de query strings. Entretanto, acessar uma página através de /controller/action?id=3&page=2 não é a nossa definição de URL amigável. No Spaghetti\*, parâmetros são geralmente passados como sendo parte da URL. Usamos dois tipos de parâmetros: parâmetros nomeados, que funcionam de maneira semelhante a query strings, e parâmetros não-nomeados, que são enviados como parâmetros para o método do controller. Vamos ver um pouco mais sobre cada um.

O tipo de parâmetro mais simples são os parâmetros não-nomeados. Eles são passados por uma URL com a estrutura /controller/action/parâmetro1/parâmetro2/.... Qualquer parte da URL após a definição do controller e da action são considerados parâmetros. Cada parâmetro não-nomeado é passado em ordem como parâmetro para o método correspondente à action.

Para receber esses parâmetros, é necessário definir os parâmetros em sua action. Por exemplo, o parâmetro mais comum recebido por uma action é o ID de um registro. Ele terá uma URL semelhante a /controller/action/12. Em nossa action, define-se o parâmetro comumente por $id:

|  |  |
| --- | --- |
| 1 | public function view($id = null) { |
| 2 | $this->set("post", $this->Posts->firstById($id)); | |

|  |  |
| --- | --- |
| 3 | } |

Recomendamos que cada parâmetro tenha um valor padrão, e que esse seja checado por sua action. Caso contrário, ocorrerão erros do PHP, ou registros vazios sendo mostrados.

Para receber o título de uma postagem como parâmetro, em vez de seu ID, é igualmente simples. Na verdade, bastaria você trocar $id por $slug, apenas para não confundir o tipo de parâmetros sendo recebidos, e sua aplicação responderia a URLs como /posts/view/titulo-de-um-post.

Você pode receber quantos parâmetros nomeados quantos forem necessários. Não é necessário se preocupar com parâmetros extra, caso hajam mais parâmetros recebidos pela URL do que aqueles definidos no método, eles apenas serão ignorados. Caso hajam menos parâmetros recebidos do que aqueles definidos no método, dependerá de seus valores padrão, e o tratamento que seu método dará a eles.

Quando faz mais sentido utilizar nomes para alguns parâmetros, o Spaghetti\* provê os parâmetros nomeados. Eles funcionam tal qual query strings, embora fiquem muito mais amigáveis na URL. A forma de um parâmetro nomeado em uma URL é /controller/action/param:value/param2:value. Para recebê-los em sua action, é necessário usar o método Controller::param() com o nome do parâmetro a ser retornado.

|  |  |
| --- | --- |
| 1 | public function view() { |
| 2 | $id = $this->param("id"); | |

|  |  |  |
| --- | --- | --- |
| 3 | $this->set("post", $this->Posts->firstById($id)); | |
| 4 | } |

O principal uso desse tipo de parâmetros é quando usamos paginação ou ordenação de registros. Por exemplo, para acessar a segunda página de uma listagem de posts ordenados por data de criação, acessaríamos a URL /posts/index/page:2/order:created/direction:desc.

Como não é possível fazer com que formulários enviados via GET sigam o mesmo padrão do Spaghetti\*, os controllers também recebem parâmetros vindos de query strings. Na verdade, para o Spaghetti\* não existe diferença entre um parâmetro nomeado e um parâmetro enviado por query string, eles são tratados da mesma maneira. Assim, tanto /page:1 e ?page=1 podem ser acessados através de $this->param("page").

## Recebendo Dados do Usuário

Além dos parâmetros recebidos pela URL, também é necessário receber os dados vindos do usuário através de formulários submetidos via POST. Nos controllers do Spaghetti\*, esses dados correspondentes à superglobal $\_POST ficam armazenados em Controller::$data.

Para detectar se uma requisição foi enviada através de um POST ou não é muito simples. Quando existem dados em $\_POST, Controller::$data é preenchido com esses dados, e fica vazio quando nenhum dado é enviado. Assim, usamos a função empty() para verificar se a requisição foi feita através de POST ou um simples GET.

|  |  |
| --- | --- |
| 1 | public function add() { |
| 2 | if(!empty($this->data)): // verificamos se há dados em $this->data | |

|  |  |  |
| --- | --- | --- |
| 3 | $this->Posts->save($this->data); // salvamos o registro com os dados vindos do usuário | |
| 4 | endif; |

|  |  |
| --- | --- |
| 5 | } |

Lembre-se: a menos que seja definido um valor id no formulário que gerou a requisição, o ID passado como parâmetro na URL não é passado para Controller::$data, sendo necessário definí-lo manualmente.

|  |  |  |
| --- | --- | --- |
| 1 | public funcion edit($id = null) { | |
| 2 | if(!empty($this->data)): |

|  |  |
| --- | --- |
| 3 | $this->Posts->id = $id; |
| 4 | $this->Posts->save($this->data); | |

|  |  |  |
| --- | --- | --- |
| 5 | endif; | |
| 6 | } |

## Models, Helpers e Components

Por padrão, o Spaghetti\* já vincula seu controller com um modelo de mesmo nome. Entretanto, você pode precisar usar um modelo diferente em determinados controllers, ou mesmo usar vários modelos dentro de um mesmo controller. Para isso, o Spaghetti\* usa a variável Controller::$uses.

|  |  |
| --- | --- |
| 1 | class UsersController extends AppController { |
| 2 | public $uses = array("Users", "Profiles"); | |

|  |  |
| --- | --- |
| 3 | } |

**Nota**: Não é necessário incluir modelos relacionados caso eles não sejam usados diretamente, o Spaghetti\* já se encarrega disso!

Caso você não deseje utilizar modelo algum em seu controller, basta definir essa variável como um array vazio:

|  |  |  |
| --- | --- | --- |
| 1 | class UsersController extends AppController { | |
| 2 | public $uses = array(); |

|  |  |
| --- | --- |
| 3 | } |

Os components e helpers desejados também devem ser incluídos no controller, através de suas respectivas variáveis Controller::$components e Controller::$helpers, assim como na definição de modelos.

|  |  |
| --- | --- |
| 1 | class UsersController extends AppController { |
| 2 | public $uses = array("Users", "Profiles"); | |

|  |  |
| --- | --- |
| 3 | public $components = array("Auth"); |
| 4 | public $helpers = array("Html", "Form", "Date"); | |

|  |  |
| --- | --- |
| 5 | } |

Por padrão, o Spaghetti\* já inclui os helpers HtmlHelper, FormHelper e PaginationHelper. Por isso, caso você utilize apenas esses helpers, não é necessária a sua definição explícita. Entretanto, caso você deseje incluir outros helpers, será necessário definí-los também.

## Usando Layouts

Fazendo o uso de [layouts](http://spaghettiphp.com.br/docs/developer-guide/views/layouts), você não precisa escrever tanto código HTML, já que é possível gerar todo o esqueleto da página apenas uma vez. Por padrão, o Spaghetti\* renderiza o layout default, que se encontra em /app/layouts/default.htm.php. Entretanto, é possível escolher qualquer outro layout disponível em sua aplicação através da variável Controller::$layout.

|  |  |  |
| --- | --- | --- |
| 1 | class HomeController extends AppController { | |
| 2 | public $layout = "home"; |

|  |  |
| --- | --- |
| 3 | } |

Muitas vezes um layout não é necessário, e você deseja que o Spaghetti\* não renderize nenhum. Nesse caso, é possível definir o valor da variável como false e pronto!

## Redirecionamentos

Após uma ação, geralmente após editar ou apagar algum registro, geralmente acontece um redirecionamento dentro da aplicação. Os controllers do Spaghetti\* já herdam o método Controller::redirect() para esse propósito. Basta definir a URL para a qual se deseja redirecionar.

|  |  |
| --- | --- |
| 1 | $this->redirect("/home"); |

Como o redirecionamento é feito através de um header HTTP, também é possível adicionar mais um header indicando o status da requisição. Basta passar como segundo parâmetro o número do status, e o Spaghetti\* se encarrega de definir o header apropriado:

|  |  |
| --- | --- |
| 1 | $this->redirect("/home", 404); // gerando um erro 404 - página não encontrada |

Além de redirecionamentos HTTP, o Spaghetti\* também suporta que você redirecione uma ação para outra do mesmo controller. Isso é feito através do método Controller::setAction(), passando o nome da ação como parâmetro. Qualquer outro parâmetro será passado diretamente para a action definida.

Esse método é bastante útil para evitar a criação de várias views para ações semelhantes, como adicionar e editar registros, por exemplo.

|  |  |
| --- | --- |
| 1 | public function add() { |
| 2 | $this->setAction("edit"); | |

|  |  |
| --- | --- |
| 3 | } |
| 4 | public function edit($id = null) { | |

|  |  |
| --- | --- |
| 5 | if(!empty($this->data)): |
| 6 | $this->data["id"] = $id; | |

|  |  |  |
| --- | --- | --- |
| 7 | $this->Posts->save($this->data); | |
| 8 | endif; |

|  |  |
| --- | --- |
| 9 | } |

## Callbacks

Os callbacks são funções executadas em determinados momentos da execução do Spaghetti\*, como antes da execução de uma ação do controller ou antes da renderização de uma view.

|  |  |
| --- | --- |
| **Callback** | **Momento da execução** |
| beforeFilter | Antes da execução da action |
| beforeRender | Depois da execução da action, mas antes da renderização da view |
| afterFilter | Após execução da action e renderização da view |

Esse callbacks podem ser usados para executar qualquer coisa que você precise para seus controllers. Para definí-los, basta criar sua função correspondente na classe de seu controller.

|  |  |  |
| --- | --- | --- |
| 1 | class HomeController extends AppController { | |
| 2 | public function beforeFilter() { |

|  |  |  |
| --- | --- | --- |
| 3 | $this->AuthComponent->check(); | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |

## AppController

AppController é o controller base para todos os outros controllers na aplicação. Isso significa que todo e qualquer método ou atributo que for definido dentro dessa classe estará disponível para todos os outros controllers da aplicação.

O uso dessa herança é muito útil no uso de callbacks, quando é necessário que eles sejam executados em todas as partes da aplicação. O componente AuthComponent, por exemplo, pode requerer que algumas configurações sejam definidas antes da execução do controller, e essas configurações precisam ser definidas em todas as partes da sua aplicação.

Por padrão, o Spaghetti\* usa o AppController da biblioteca padrão. Entretanto, você pode criar o arquivo /app/controllers/app\_controller.php, e então o Spaghetti\* passará a usá-lo para estender todos os outros controllers de sua aplicação.