# Introdução

Desenvolver um sistema com qualidade, dentro do prazo e custo estipulados é um desafio que as empresas de desenvolvimento de *software* sempre buscaram atender. Uma solução conhecida para este problema é a reusabilidade.

Esta consiste em reaproveitar partes do sistema, que já existem e desta forma obter maior produtividade e qualidade no desenvolvimento. O ganho de qualidade acontece em virtude da reutilização de partes que já foram testadas previamente em outros trechos do *software*, ou até mesmo em outras aplicações. O aumento da produtividade decorre do reaproveitamento de partes já existentes.

A reusabilidade pode ser feita através de cópia de código-fonte, o que não é recomendado, pois replicar o código-fonte pode implicar em problemas de manutenção. Por meio de classes, componentes e herança, podendo chegar a um nível mais alto que são os *frameworks*.

Um *framework* é uma infra-estrutura ou esqueleto de uma família de aplicações projetado para ser reutilizado. Basicamente, aplicações específicas são construídas especializando as classes do *framework* para fornecer a implementação de alguns métodos e, assim a aplicação implementa somente as funcionalidades específicas (DEUTSCH, 1989).

As vantagens do uso de *frameworks* nos projetos, de acordo com Assis (2003) e Sauvé (2004) são:

- Redução do tempo de codificação: muitas funcionalidades necessárias já estão disponíveis no *framework*;
- Soluções bem testadas por outras pessoas: Os *frameworks* são utilizados por muitas pessoas, isto garante um alto nível de maturidade ao se descobrir erros e adicionar novas funcionalidades;
- Programadores implementam somente o que é necessário: não é preciso que se codifique todo o *software*, pois se utiliza os componentes que já estão prontos;
- Redução de erros: os *frameworks* diminuem o número de linhas de codificação, desta forma, reduzem também a possibilidade de erros comuns.

É importante destacar a diferença de um *framework* para uma biblioteca de classes, dentre as diferenças a de maior destaque é a forma como o fluxo de execução é controlado. Quando se utiliza uma biblioteca de classes, é o programador que define como vai ser a execução dos comandos. Diferentemente das bibliotecas, os *frameworks* invertem este controle de execução, sendo eles os responsáveis por controlar o fluxo da execução.

# Sobre o CakePHP

CakePHP é um *framework* para desenvolvimento ágil, escrito na linguagem PHP. Ele usa padrões de projetos conhecidos, como ORM e MVC; utiliza também o paradigma das convenções no lugar do uso de configurações. Este *framework* busca a redução do custo de desenvolvimento através da diminuição do número de linhas de código-fonte.

Ele é um *software* livre, licenciado sob a licença MIT - http://www.opensource.org/licenses/mit-license.php.

# Vantagens do CakePHP

Dentre os diversos recursos que agregam valor ao utilizar o CakePHP, podemos citar:

- *Software* livre com uma comunidade crescente e ativa.
- Tem uma licença flexível (MIT).
- Compatível com PHP 4 e 5.
- Facilidade de gerar a codificação do CRUD.
- *Scaffolding* - geração de telas dinâmicas (*on the fly*).
- Geração de código-fonte
- Arquitetura MVC
- Uso de URLs amigáveis
- Validação embutida

- Templates rápidos e flexiveis, utilizando sintaxe PHP
- Desenvolvimento de *views* com *helpers*
- Diversos componentes disponíveis
- Lista de controle de acessos (ACL)
- Proteção contra dados maliciosos (*Data Sanitization*)
- Ferramentas para *caching*
- Internacionalização e Localização
- Funciona em qualquer servidor *web* rodando PHP.
- E o principal: devolve ao programador a diversão de programar.

# 1 – Instalação

## 1.1 – Requisitos para Utilização do CakePHP

O *framework* CakePHP funciona em qualquer servidor web que seja capaz de executar *scripts* em linguagem PHP 4 ou superior.

- Servidor web: Apache 2.0, LightHTTP, IIS. Sendo recomendado a utilização do apache com o módulo mod_rewrite.
- PHP 5.2.6 ou superior. O CakePHP também tem suporte para versão 4 PHP.
- Banco de Dados suportados pelo CakePHP:
    - MySQL (4 ou superior), PostgreSQL, Firebird, MS SQL SERVER, Oracle, SQLite, ODBC, ADOdb

## 1.2 – Configurando o Apache

O servidor web Apache é um *software* livre mantido pela organização Apache Foundation. O servidor Apache é o servidor mais utilizado para hospedar páginas na internet.

O CakePHP pode ser configurado de diversas maneiras no servidor Apache, este capítulo irá apresentar as diferentes formas de configuração, destacando seus prós e contras.

O arquivo de configuração do Apache é o **httpd.conf**. Este arquivo é encontrado nos sistemas *nix dentro do diretório /etc/apache2/. No windows o arquivo será encontrado dentro do diretório "conf", geralmente localizado no diretório de instalação do Apache.

### 1.2.1 – Configurando o mod_rewrite

Este módulo é utilizado para criar endereços (URLs) mais amigáveis, evitando endereços longos com diversos &. Por exemplo:

**Nota**

O endereço: http://www.dominio.com.br/index.php?nome=teste&data=20081110. Utilizando a reescrita de URL poderia ser escrito assim: http://www.dominio.com.br/teste/20081110

Para habilitar o módulo *rewrite* no servidor apache basta descomentar* a seguinte linha:

```
#LoadModule rewrite_module modules/mod_rewrite.so
```

Observação: O caminho do módulo pode alterar dependendo da instalação do Apache ou do sistema operacional.

## 1.3 – Instalando o CakePHP

O primeiro passo para a instalação do *framework* é fazer o seu *download* no endereço:

- Página principal do projeto existe um link para a última versão estável:

```
http://cakephp.org
```

- Página contendo o projeto do CakePHP e diversos outros projetos (components, plugins, etc):

```
http://cakeforge.org/projects/cakephp/
```

Após o *download* dos fontes, é necessário descompactar o arquivo no *DocumentRoot* do apache.

Pasta da DocumentRoot: *C:\Arquivos de Prgramas\...\htdocs*

Sendo o conteúdo de *C:\Arquivos de Prgramas\...\htdocs* as pastas *app*, *cake* e *vendors*.

## 1.4 – Estrutura de diretórios

Após descompactar o arquivo baixado pelo site do CakePHP, a seguinte estrutura é encontrada:

| Diretório | Descrição |
|---|---|
| /app/ | Aqui a aplicação é desenvolvida. |
| /app/config/ | Onde ficam os arquivos de configuração.<br><br>O arquivo de configuração de banco de dados (database.php) se encontra aqui. |
| /app/models | As classes da camada de modelo devem ser escritas neste diretório. |
| /app/controllers | As classes da camada de controle devem ser escritas neste diretório. |
| /app/views | Os arquivos da camada de visão devem ser escritas neste diretório. |
| /app/webroot/ | Todas as requisições são direcionadas para este diretório, onde o *Dispatcher* trata quais arquivos são necessários para atender cada requisição.<br><br>As pastas deste diretório servem como abrigo para arquivos css, imagens, javascripts e qualquer outro arquivo que precisa estar disponíveis para requisição direta. |
| /cake/ | Os arquivos do *framework* ficam neste diretório.<br><br>O desenvolvedor não deve alterar o conteúdo desta pasta, somente se souber o que está fazendo. |

Estrutura de diretórios do CakePHP

## 1.5 – Arquivos de configuração do CakePHP

O CakePHP possui poucos arquivos de configuração, devido sua arquitetura baseada em convenções.

### 1.5.1 – O arquivo de configuração `database.php`

Para iniciar o desenvolvimento de uma aplicação, é necessário apenas configurar o banco de dados por meio do arquivo: *diretório_do_cake/app/conf/database.php*.

O banco de dados é configurado pelo array: `$default`, que é ilustrado abaixo.

```
var $default = array(
        'driver'      => 'mysql',
        'persistent'  => false,
        'host'        => 'localhost',
        'login'       => 'cakephpuser',
        'password'    => 'c4k3roxx!',
        'database'    => 'my_cakephp_project',
        'prefix'      => ''
    );
```

Configuração do arquivo `database.php`

### 1.5.2 – O arquivo de configuração `core.php`

O arquivo `core.php` possui as configurações básicas do *framework*. Confira abaixo as suas opções de configuração:

| Variável | Descrição |
|---|---|
| debug | Configura o nível de mensagens de debug que será mostrado na tela.<br><br>**0** = Modo sem nenhuma mensagem, para uso do sistema em produção.<br>**1** = Mostra mensagens de erros e advertências.<br>**2** = Mostra mensagens de erros, advertências e comandos SQL.<br>**3** = Mostra mensagens de erros, advertências, comandos SQL e variáveis do controle. |
| App.baseUrl | Descomente esta linha caso deseje utilizar o CakePHP sem o mod_rewrite.<br>É importante apagar todos os arquivos .htaccess<br>No IIS pode ser necessário colocar uma ? concatenada, como demonstrado abaixo:<br>Configure::write('App.baseUrl', env('SCRIPT_NAME')."?"); |
| Security.salt | Uma string aleatória, usada para a tabela *hash* de segurança. |

Configurações do *framework* no arquivo `core.php`

# 2 – Desenvolvendo com o CakePHP

## 2.1 – Convenções no CakePHP

Apesar de tomar um pouco de tempo para aprender as convenções do CakePHP, você ganha tempo em um longo processo: seguindo as convenções, você ganha funcionalidades gratuitamente e livra-se de madrugadas de manutenção de arquivos de configuração. Convenções também fazem com que o sistema fique uniformemente desenvolvido, permitindo que outros desenvolvedores o ajudem mais facilmente.

As convenções no CakePHP têm sido produzidas por anos de experiência em desenvolvimento web e boas práticas. Apesar de sugerimos que você use essas convenções enquanto desenvolve em CakePHP, devemos mencionar que muitos desses princípios são facilmente sobrescritos – algo que especialmente acontece quando trabalha-se com sistemas legados.

### 2.1.1 – Todas as Convenções

| Local | Regra | Exemplo |
|---|---|---|
| Banco de dados | Nome das tabelas: plural, minúsculo. | usuarios, posts, artigos |
| Banco de dados | Chave primária: nome id, tipo int, auto_increment | id int(11) |
| Banco de dados | Nome dos campos: minúsculo, usando sublinhados. | nome, rua, data_de_nascimento |
| Banco de dados | Chave estrangeira: nome da tabela relacionada no singular, acrescentando sublinhado e id. | cliente_id,grupo_id |
| Banco de dados | Relacionamento N-M (HABTM), regra para a tabela de ligação: nome das tabelas em ordem alfabética, separadas por sublinhado. | produtos_usuarios, grupos_usuarios |
| Modelo | Nome do arquivo: singular e CamelCase | Usuario.php, Grupo.php, Cliente.php |
| Modelo | Nome da classe: singular e CamelCase | Usuario, Grupo |
| Controle | Nome do arquivo: plural, sublinhados e terminado com Controller | usuarios_controller.php, grupos_controller.php |
| Controle | Nome da classe: plural, CamelCase e terminado com Controller | UsuariosController, GruposController |
| Visão | Local do arquivo: dentro da pasta views/nome_do_controle/ | views/usuarios/, views/grupos/ |
| Visão | Nome do arquivo: deve ter o nome da ação do controle, com sublinhados. | views/usuarios/index.ctp, views/grupos/add.ctp |

Lista de todas as convenções do CakePHP

## 2.3 - A Arquitetura MVC

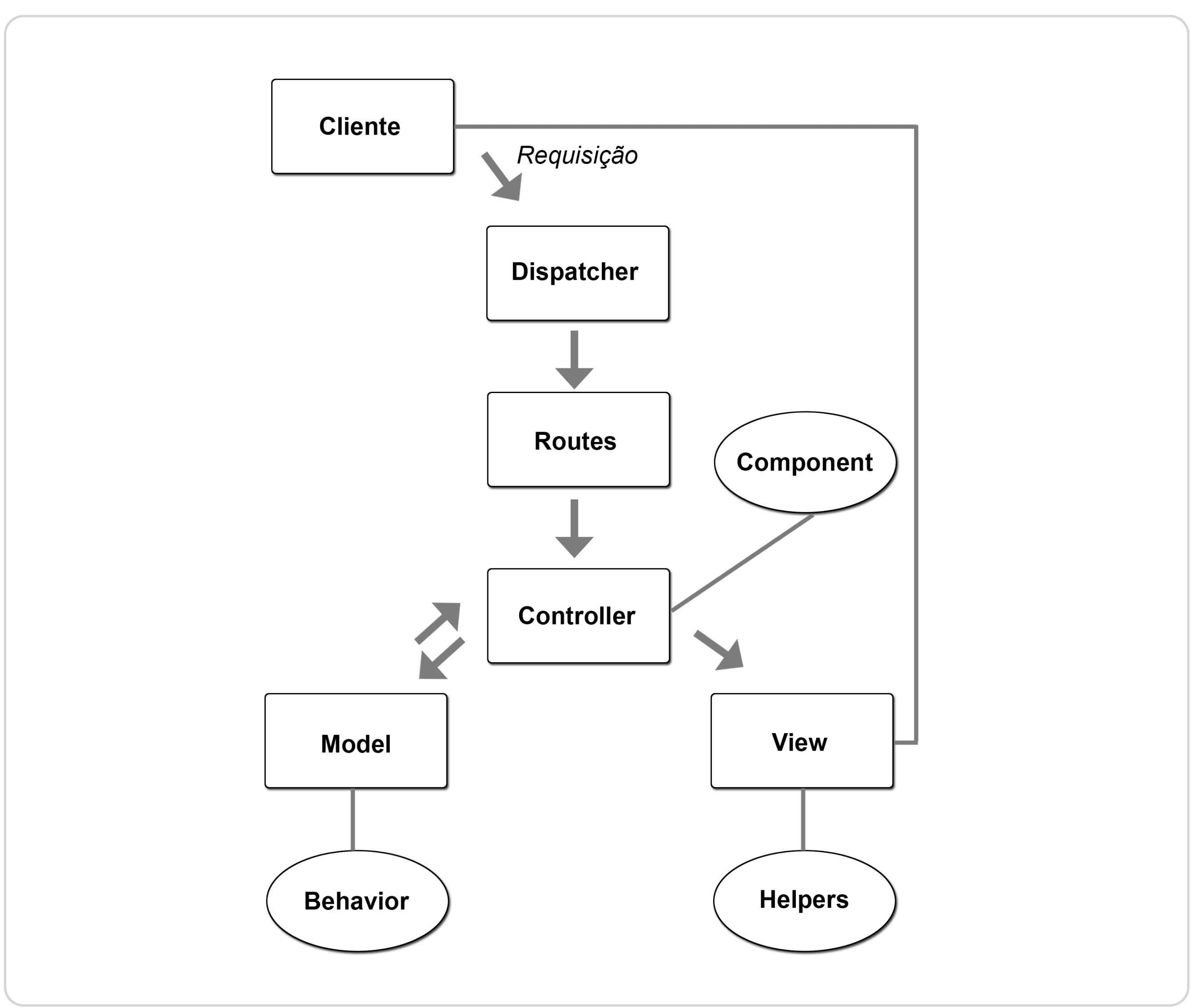
A medida que os *softwares* foram ficando maiores e mais complexos, tornou-se necessário organizar o código-fonte de forma a facilitar o seu entendimento.

Uma solução criada para este problema foi a divisão da aplicação em 3 camadas com funcionalidades distintas, está solução é denominada arquitetura MVC (Model - View - Controller).

Basicamente a arquitetura MVC separa o desenvolvimento da aplicação em:

- Camada de Modelo: parte da programação destinada ao tratamento dos dados.
- Camada de Controle: responsável pelas regras de negócio
- Camada de Visão: responsável pelo o que é apresentado para o cliente

A seguir, é apresentado o fluxo de execução de uma requisição feita ao CakePHP.



Fluxo da execução de uma requisição no CakePHP

## 2.3.1 - A camada de modelo

Essa camada é responsável por representar dados. Ela é utilizada para fornecer acesso aos dados, que podem estar armazenados em um banco de dados, um arquivo texto, um arquivo CSV, dentre outras possibilidades. De forma mais usual, geralmente um modelo específico representa uma tabela de um banco de dados.

Em programação orientada a objeto podemos dizer que um modelo é um objeto que representa algo da realidade. Em outras palavras, um modelo pode ser a representação de uma televisão ou uma cama ou qualquer outro objeto. Para se definir um modelo no CakePHP deve se fazer o seguinte:

```
class Cliente extends AppModel {
      var $name = 'Cliente';
}
```

Como se pode ver, neste exemplo, a classe Cliente herda da classe AppModel que por sua vez herda da classe Model. A princípio a classe AppModel está vazia. Ela existe para que seja possível gerar determinados comportamentos comuns a todos os modelos.

### 2.3.1.1 - Atributos dos modelos

#### 2.3.1.1.1 - useDbConfig

A propriedade *useDbConfig* define, através de uma *string*, o nome da conexão com o banco de dados. Por padrão a conexão que o CakePHP usa é definida no arquivo app/config/database.php e tem o valor *default*. É necessário criar o array no arquivo `database.php` com os dados da conexão. Veja o exemplo a seguir:

```
class Cliente extends AppModel {

	var $name = 'Cliente';
	var $useDbConfig = 'teste';

	var $useTable = false;
	//ou
	var $useTable = 'testes';

	var $tablePrefix = 'user_';
	var $primaryKey = 'email';
	var $displayField = 'nome';

	var $order = "Cliente.nome DESC";
	//ou
	var $order = array("Cliente.nome" => "asc", "Cliente.email" => "DESC");
}
```

Código no arquivo do modelo

#### 2.3.1.1.2 - useTable

Este atributo define o nome da tabela que o modelo utilizará, somente existirá a necessidade de atribuir um valor, caso não esteja sendo utilizada as convenções. Se o modelo não necessitar usar tabela alguma, pode-se utilizar essa propriedade com o valor false.

#### 2.3.1.1.3 - tablePrefix

Este atributo é utilizado para definir o prefixo de alguma tabela. Por padrão nenhum prefixo é definido. Pode-se também definir o prefixo no arquivo `app/config/database.php`, entretanto se for usado tablePrefix no model, essas definições sobrescrevem as definições definidas no `app/config/database.php`.

#### 2.3.1.1.4 - primaryKey

Usualmente toda tabela possui o campo id definido como chave primária. Entretanto, existem casos onde a tabela pode não possuir um campo id, ou pretende-se usar outro campo como chave primária. Neste caso usamos o atributo primaryKey para definir qual campo funcionará dessa forma.

#### 2.3.1.1.5 - displayField

O displayField define qual campo da tabela será usado como rórulo (label). Esse rótulo é usado quando se utiliza scaffolding e find('list'). Por padrão a camada de modelo usa os campos name ou title. Não se pode usar mais que um valor para displayField. Portanto, não funcionará utilizar var $displayField = array('valor1', 'valor2').

#### 2.3.1.1.6 - recursive

Recursive é utilizado quando o modelo possui relacionamento com outros modelos. Através dessa propriedade

pode-se definir até onde o modelo deve trazer os relacionamentos quando existem chamadas para as funções find e read.

Como exemplo prático, imagine uma aplicação onde uma universidade possui diversos cursos, estes pussuem disciplinas, que têm professores. Logo, Universidade HasMany Curso; Curso HasMany Disciplina; Disciplina HasMany Professor. Ao fazer uma busca na entidade Curso ($this->Curso->Find()), a busca pode se comportar das seguintes formas:

| Profundidade | Descrição |
|---|---|
| -1 | Cake procura somente os cursos. |
| 0 | Cake procura por cursos e a universidade ao qual os cursos pertencem. |
| 1 | Cake procura por cursos, universidades e disciplinas associados. |
| 2 | Cake procura por cursos, universidade, disciplinas associados e os professores associados a disciplina. |

Valores possíveis para a variável `recursive`

#### 2.3.1.1.7 - order

Define qual a ordem que os resultados das pesquisas feitas com find vão retornar. Pode ter o valor ASC, DESC, pode-se também ordenar por um campo, ou por um Modelo.campo. É possível ainda utilizar um array com mais de um critério de ordenação.

#### 2.3.1.1.8 - data

Neste atributo é armazenado todos os dados retornados em uma pesquisa. Esses dados são armazenados em forma de arrays multidimensionais, que possuem os nomes dos campos como índices dos arrays.

#### 2.3.1.1.9 - validate

Através deste atributo é possível ao modelo fazer validações antes de armazenar os dados no banco de dados ou onde estes dados forem armazenados.

```
class Teste extends AppModel {
	var $validate = array(
		'nome' => array(
			'rule'=>array('between',2,20),
			'message'=>'O campo deve possuir de 2 a 20 caracteres.
		),
		'email' => array(
			'rule'=>array('email'),
			'message'=>'O campo deve conter e-mail válido.'
		),
		'comentario' => array(
			'rule'=>array('between',5,500),
			'message'=>'O campo deve possuir de 5 a 500 caracteres.'
```

```
            ),
      );
}
```

Exemplo para uma validação com uma regra:

Os parâmetros do array validate

| Chave | Descrição |
|---|---|
| Rule | Regra de válidação do campo |
| Required | O campo precisa ter dados |
| allowEmpty | O campo aceita valor vazio |
| on | Define em qual lugar a validação deve ser feita. 'on'=>'create' ou 'on'=>'update' |
| message | Define a mensagem que será apresentada, caso o campo não passe na validação |

Parâmetros do *validate*

Algumas regras padrões de validação do CakePHP

| Regra | Validação |
|---|---|
| alphaNumeric | Para o campo ser válido ele deve conter apenas caracteres alfanuméricos.<br>**array('login' => array('rule' => 'alphaNumeric'));** |
| between | O campo pode ter qualquer caracter, mas o tamanho do conteúdo do campo estar entre a faixa determinada.<br>**array('senha' => array('rule' => array('between',6,8)));** |
| date | O campo precisa ser uma data válida.<br>**array('data_inicio' => array('rule' => 'date'));** |
| email | O campo precisa ser um email.<br>**array('email' => array('rule' => 'email'));** |
| isUnique | O campo deve ser único no banco de dados.<br>**array('flag' => array('rule' => 'isUnique'));** |
| numeric | O campo precisa ser um número válido.<br>**array('valor' => array('rule' => 'numeric'));** |

Regras padrões de validação do CakePHP

#### 2.3.1.1.9.1 – Validação Avançada

Usando uma regra customizada.

```
var $validate = array(
	'login' => array(
		'rule' => array('custom', '/^[a-z0-9]{3,}$/i'),
		'message' => 'Somente letras e números são aceitos, mínimo de 3 
caracteres'
	)
);
```

Exemplo de uma regra customizada utilizando expressão regular

```
var $validate = array(
	'codigoPromocional' => array(
		'rule' => array('limiteDeUso', 250),
		'message' => 'Código informado foi usado mais vezes do que o permitido.'
	)
);
function limiteDeUso($data, $limite){
	$contador = $this->find('count', array('conditions' => $data, 'recursive' => -1));
	return $contador < $limit;
}
```

Exemplo de uma regra customizada utilizando uma função própria

```
var $validate = array(
	'login' => array(
		'alphanumeric' => array(
			'rule' => 'alphaNumeric',
			'message' => 'Somente alfanuméricos são aceitos'
		),
		'between' => array(
			'rule' => array('between', 4,8),
			'message' => 'O campo deve possuir entre 4 e 8 caracteres'
		),
		'isUnique' => array(
			'rule' => array('isUnique'),
			'message' => 'O login já existe em nossa base de dados'
		),
	)
);
```

Usando mais de uma validação no mesmo campo

#### 2.3.1.1.10 - name

Este atributo permite que o código do CakePHP na camada de modelo seja compatível com o php4. Portanto, é recomendável sempre utilizar esta declaração já que pode ocorrer da aplicação ter que mudar para um servidor que utiliza o php nessa versão.

### 2.3.1.2 - Associando os modelos

#### 2.3.1.2.1 - hasOne

Esse tipo de relacionamento é usado quando uma tabela pode possuir apenas um resultado relacionado em

outra tabela. Por exemplo, em uma aplicação de comunidade virtual onde se tem uma tabela de Usuário e outra de Perfil do Usuário. Cada usuário pode ter apenas um perfil. Neste caso o relacionamento entre a tabela Usuário e Perfil de Usuário é do tipo hasOne.

```
class Usuario extends AppModel {
	var $name = 'Usuario';
	var $hasOne = array('Perfil');
}
```

Exemplo de uso do hasOne

O exemplo acima é a forma mais simples de se estabelecer o relacionamento do tipo hasOne. Existem casos onde é necessário ter mais controle sobre o relacionamento estabelecido. Veja o exemplo seguinte:

```
class Usuario extends AppModel {
	var $name = 'Usuario';

	var $hasOne = array(
		'Perfil' => array(
			'className'	=> 'Perfil',
			'conditions'	=> array('Perfil.publicado' => '1'),
			'dependent'	=> true
		)
	);

}
```

Exemplo de uso avançado do hasOne

#### 2.3.1.2.2 – hasMany

Seguindo nosso exemplo, o Usuário de nossa comunidade virtual pode ter diversos artigos. Portanto a tabela Usuário estabelece um relacionamento do tipo hasMany com a tabela Artigos.

```
class Usuario extends AppModel {
	var $name = 'Usuario';
	var $hasMany = array('Artigo');
}
```

Exemplo de uso de hasMany

#### 2.3.1.2.3 – belongsTo

Depois de estabelecido o relacionamento do tipo HasOne ou HasMany podemos estabelecer o relacionamento belongsTo na tabela relacionada. Esse passo pode ser importante se pretendemos recuperar dados a partir dos registros da tabela que é relacionada à tabela que estabeleceu o hasOne ou HasMany. Seguindo o exemplo exposto em hasOne, a tabela Perfil do Usuário pertence a tabela Usuário. Ou seja, um perfil inscrito na tabela Perfil do Usuário pertence a um usuário pertencente à tabela Usuário.

```
class Perfil extends AppModel {
	var $name = 'Perfil';
	var $belongsTo = array('Usuario');
}
```

Exemplo de uso de belongsTo

#### 2.3.1.2.4 – hasAndBelongsToMany (HABTM)

Quando um modelo pode ter diversos resultados associados de outro modelo e vice-versa é necessário utilizar o relacionamento do tipo hasAndBelongsToMany. Neste caso, será necessário criar uma tabela intermediária na estrutura do banco de dados. O nome dessa nova tabela deverá conter o nome das duas tabelas relacionadas em ordem alfabética e separadas por *underscore* (_).

O nome das chaves estrangeiras deve seguir o padrão do CakePHP. Em nosso exemplo, um usuário pode pertencer em diversas comunidades. E, ao mesmo tempo, uma comunidade pode possuir diversos usuários. Então o nome da nossa tabela seria comunidades_usuarios e o nome dos campos na tabela intermediária seria respectivamente id, comunidade_id e usuario_id.

```
class Usuario extends AppModel {
	var $name = 'Usuario';
	var $hasAndBelongsToMany = array(
		'Comunidade' => array(
			'className' => 'Comunidade',
			'joinTable' => 'usuarios_comunidades',
			'foreignKey' => 'usuario_id',
			'associationForeignKey' => 'comunidade_id',
			'unique' => true
		)
	);
}
```

Exemplo de hasAndBelongsToMany

### 2.3.1.3 – Recuperando dados

#### 2.3.1.3.1 – find

**find($type, $params)**

**$type** – pode receber os valores 'all', 'first', 'list', 'neighbors', 'count' ou 'threaded'. O valor padrão utilizado é o 'first'.

**$params** – trata-se de um array que pode possuir os seguintes parâmetros:

```
array(
	//array de condições
	'conditions' => array('Modelo.campo' => $thisValue),

	//valor inteiro
	'recursive' => 1,

	 //array contendo o nome dos campos da tabela a ser recuperadas
	'fields' => array('Modelo.campo1', 'Modelo.campo2'),

	//string ou array que define a ordem dos resultados
	'order' => array('Modelo.created', 'Modelo.campo3 DESC'),

	//campos para serem usados no GROUP BY
	'group' => array('Modelo.campo'),
```

```
        //valor inteiro
        'limit' => n,

        //valor inteiro
        'page' => n,
)
```

Parâmetros do array $params

Se estiver sendo usado $type do tipo list, o campo $key no array $params irá definir o índice e o valor.

```
// gera uma lista contendo Usuario.email como índice e Usuario.nome como valor
$this->Post->find('list', array('fields'=>array('Usuario.email',
'Usuario.nome')));
```

Exemplos do $type *list*

#### 2.3.1.3.2 – findAllBy

**findAllBy<fieldName>(string $value)**

Através deste método é possível procurar por certo campo da tabela. Para isso basta adicionar o nome do campo no formato *CamelCase* logo após o findAllBy. O parâmetro $value deve ser uma *string* com o critério da busca.

```
$this->Usuario->findAllByComunidade('Hackers');
```

#### 2.3.1.3.3 – query

Através desse método do Modelo é possível fazer requisições ao banco de dados de forma customizada. Neste caso é recomendável utilizar a biblioteca Sanitize para prevenir ataques do tipo injection e cross-site scripting.

```
$this->Produto->query("SELECT preco, cor FROM produto LIMIT 100;");
```

### 2.3.1.4 – Salvando dados

#### 2.3.1.4.1 – save

**save(array $data = null, boolean $validate = true, array $fieldList = array())**

**$data:** array contendo os dados a serem salvos, tendo como índice os nomes dos campos onde estes dados deverão ser salvos.

**$boolean:** se for atribuido o valor *true*, o método save irá validar os dados antes de salvá-lo. As regras de validação podem ser montadas no array $validate no próprio Modelo.

**$fieldList:** os campos onde os dados deverão ser salvos.

Depois que os dados foram salvos, pode-se acessar o valor do id pode ser recuperado no atributo $id do objeto do modelo.

#### 2.3.1.4.2 – saveAll

**saveAll(array $data = null, array $options = array())**

Esse método pode ser utilizado para salvar dados de um modelo ou, além de salvar os dados do modelo, gravar os dados nos seus modelos relacionados.

No parâmetro *$options* pode ser usado os seguintes opções:

| Chave | Descrição |
|---|---|
| validate | Se atribuido *false* não ocorre validação. Se atribuído *true*, cada registro é validado antes de serem salvos. Se atribuído *first* todos os dados são validados antes de ser salvo. Se for atribuído *only* os dados são apenas validados e não salvos. |
| atomic | Se for atribuído *true* o CakePHP irá tentar salvar todos os registros em uma transação única. Se o banco de dados não der suporte a transação única, deverá ser usado false. O valor padrão é true. |
| fieldList | array com o nome dos campos onde os dados serão salvos |

Chaves do parâmetro `$options`

#### 2.3.1.4.3 – create

**create(array $data = array())**

Esse método é utilizado para gravar um novo registro. Se, ao pedir para criar um novo registro com esse método, o modelo estiver com o parâmetro $data populado com dados, então estes serão utilizados para criar o novo registro.

### 2.3.1.5 – Apagando dados

#### 2.3.1.5.1 – del

**del(int $id = null, boolean $cascade = true);**

Apaga o registro identificado pelo parâmetro $id. Por padrão apaga os registros relacionados a ele. Se uma categoria de produto for apagada, por padrão o CakePHP irá apagar todos os produtos relacionados à essa categoria.

#### 2.3.1.5.2 – deleteAll

**deleteAll(mixed $conditions, $cascade = true, $callbacks = false)**

Funciona igual a del e remove. A única diferença é que pode-se estabelecer condições, no formato de fragmentos de uma query, no parâmetro $conditions.

### 2.3.1.6 – Callbacks

#### 2.3.1.6.1 – beforeFind

**beforeFind(mixed $queryData)**

Esse *callback* é chamado antes que alguma operação de find seja chamada. O parâmetro *$queryData* contém informações sobre a *query* atual (condições, campos, etc.). Se este método retornar *false* a operação de busca não irá ocorrer.

#### 2.3.1.6.2 - afterFind

**afterFind(array $results, bool $primary)**

Esse *callback* é utilizado para introduzir lógica após uma operação de find. É importante para tratar dados como alterar datas ou valores retornados pela busca. Os resultados da busca podem ser acessados através do parâmetro $results. Este método deverá retornar o resultado da busca com os valores alterados.

#### 2.3.1.6.3 - beforeValidate

**beforeValidate()**

Esse callback é utilizado para alterar alguma validação ou inserir lógica antes da validação ocorrer. Se esse *callback* retornar *false* a validação não irá ocorrer.

#### 2.3.1.6.4 - beforeSave

**beforeSave()**

Esse *callback* é utilizado para inserir lógica antes do dados serem salvos e logo após os dados serem validados. É importante para alterar algum formato de data que precise ser armazenado de forma diferente no banco de dados.

#### 2.3.1.6.5 - afterSave

**afterSave(boolean $created)**

*Callback* utilizado para inserir lógica depois que um registro foi salvo. O parâmetro *$created* irá ser *true* se um novo registro foi criado, será false se tiver ocorrido uma operação de atualização.

#### 2.3.1.6.6 - beforeDelete

**beforeDelete()**

Se for necessário inserir alguma lógica antes de apagar um registro, esta deve ser posta neste *callback*. Ela deverá retornar o valor *true* se pretende-se que a deleção ocorra e *false* em caso contrário.

#### 2.3.1.6.7 - afterDelete

**afterDelete()**

Se for necessário inserir qualquer lógica depois de apagar um registro, esta lógica deve ser inserida nesse *callback*.

### 2.3.2 - A camada de controle

A camada de controle é usada para gerenciar a lógica da sua aplicação. Na maioria das vezes um controle irá gerenciar a lógica de um modelo. Os arquivos de controle devem ser salvos dentro do diretório *app/controllers/*.

Todo controle herda da classe *appController* que por sua vez herda da classe *Controller*. A classe *appController* encontra-se no arquivo `app/app_controller.php`. Os métodos criados nesta classe ficam disponíveis para todos os controles da aplicação. A classe *Controller*, por sua vez, é uma biblioteca padrão do CakePHP.

```
class UsuarioController extends AppController {
      function edit($id = null) {
```

```
                //ações desse método vem aqui
    }
      function delete($id) {
                //ações desse método vem aqui
      }
}
```

**2.3.2.1 – Atributos da camada de Controle**

**2.3.2.1.1 – name**

Este parâmetro deve se usado para garantir que a aplicação seja compatível com php versão 4. O valor de *$name* deve ser igual ao nome do controle.

```
      var $name = 'Usuarios';
      var $components = array('Email');
      var $helpers = array('Html','Form','Ajax');
      var $uses = array('Usuarios', 'Perfis');
      var $layout = 'default';
```

**2.3.2.1.2 – components**

Esse atributo é utilizado quando se precisa usar um componente no controle. Os componentes são bibliotecas do CakePHP que trazem funcionalidades à camada de controle.

**2.3.2.1.3 – helpers**

Esse atributo é utilizado quando se precisa usar um helper no controle. Geralmente helpers são bibliotecas do CakePHP que trazem funcionalidades na camada de visão. Embora sua importância resida na camada de visão, eles devem ser importados na camada de controle.

**2.3.2.1.4 – uses**

As vezes existe a necessidade de se usar outros modelos que não aquele relacionado com o *controller*. Nestes casos, é necessário utilizar esse atributo para selecionar um ou mais modelos diferentes.

**2.3.2.1.5 – layout**

Esse atributo é utilizado para selecionar qual *view* (tela de apresentação gerada na camada de visão) será utilizada. Neste caso, o CakePHP irá procurar no diretório */app/views/layouts* pelo arquivo atribuído a esse atributo. O nome que deve ser atribuido a *layout* deverá ser o nome do arquivo menos a sua extensão, no caso *.ctp*. Caso não seja utilizado esse atributo, o CakePHP irá procurar por */app/views/layouts/default.ctp*.

**2.3.2.1.6 – pageTitle**

Esse atributo é utilizado para trocar o título da página. Entenda-se como título da página o texto que é utilizado entre as *tags title* na linguagem de marcação de texto.

```
class NoticiasController extends AppController {
      function apresenta() {
                $this->pageTitle = 'Notícias';
```

```
    }
}
```

#### 2.3.2.1.7 - params

**$this->params**

Esse atributo fornece acesso aos parametros da requisição atual. É muito utilizado para obter informações sobre informações que foram enviadas através dos métodos *POST* ou *GET*.

#### 2.3.2.1.8 - data

**$this->data**

Armazena os dados enviadas da camada de visão para a camada de controle através do *FormHelper*.

```
class UsuariosController extends AppController {
	function apresenta() {
		if(isset($this->data['Usuario']['nome'])) {
			echo $this->data['Usuario']['nome'];
		}
	}
}
```

### 2.3.2.2 - Métodos da camada de Controle

#### 2.3.2.2.1 - set

**set(string $var, mixed $value)**

Através desse método é possível enviar dados para a camada de visão. Também é possível passar um array como parâmetro. Por exemplo: $this->set($nomes).

```
$this->set('usuarios', $this->Usuario->find('all'));
```

#### 2.3.2.2.2 - render

**render(string $action, string $layout, string $file)**

Esse método sempre é chamado no fim de todo método de um controle. Através dele é definido qual arquivo de visão será embutido dentro do layout. Se *render* não é utilizado explicitamene no métdo do controle, então o controle irá procurar por uma tela com mesmo nome da ação do controle. Por exemplo, no método list do controle usuario o render por padrão iria procurar pelo arquivo */app/views/usuario/list.ctp*.

É possível especificar outra visão através de *$action*.

Se *$this->autoRender* estiver atribuído com o valor *false*, no fim do controle não irá ser executado esse método. Se pretende-se ler um elemento, deve-se escrever o seguinte código: *$this->render('/elements/ajax')*, que irá procurar pelo arquivo *app/views/elements/ajax.ctp*.

#### 2.3.2.2.3 - redirect

**redirect(string $url, integer $status, boolean $exit)**

Utilizado para redirecionar o usuário para outra *view*. O parâmetro *$url* deve conter um caminho relativo no padrão de url do CakePHP. O parâmetro *$status* permite definir o estado da requisição HTTP. Pode-se, por exemplo, atribuir o valor 404, que corresponde a erro na busca do arquivo. Se for atribuído o valor *false* ao parâmetro *exit*, o método não vai executar a função *exit()* após executada.

```
if($success) {
	$this->redirect(array('controller' => 'vendas', 'action' => 'sucesso'));
} else {
	$this->redirect(array('controller' => 'vendas', 'action' => 'insucesso'));
}
```

#### 2.3.2.2.4 - flash

**flash(string $message, string $url, integer $pause)**

O método *flash* é similar ao *redirect*. A única diferença é a possibilidade de mostrar uma mensagem ao usuário antes dele ser redirecionado à página. No parâmetro *$message* deve-se atribuir a mensagem que deseja-se mostrar ao usuário antes que este seja redirecionado para a *url* definida no parâmetro *$url*. No parâmetro *$pause* define-se o tempo (em segundos) de exposição da mensagem.

#### 2.3.2.2.5 - referer

Retorna a url que foi utilizada para chegar à requisição atual. Veja o exemplo a seguir:

```
class UsuariosController extends AppController {
	function admin_view($id = null) {
		echo $this->referer();
	}
}
```

#### 2.3.2.2.6 - paginate

Esse método é utilizado para gerar paginação dos resultados. Pode-se utilizar as chaves conditions, fields, order, limit. Da mesma forma como se faz no método find.

```
class NoticiasController extends AppController {
	var $paginate = array(
		'conditions' => 'Noticia.canal_id = 2'
		'fields' => array('Noticia.titulo', 'Noticia.manchete'),
		'limit' => 4,
		'order' => array(
			'Noticia.criada' => 'asc'
		)
	);
}
```

### 2.3.2.3 - Callbacks

#### 2.3.2.3.1 - beforeFilter

Esta função é executada antes de qualquer ação do *controller*. É importante utilizá-la para inspecionar a existência de alguma variável de sessão ou verificar se o usuário possui permissões de acesso.

#### 2.3.2.3.2 - beforeRender

Esta função é executada depois da ação do *controller* e antes da visão ser renderizada.

#### 2.3.2.3.3 - afterFilter

Este *callback* é chamado depois que uma ação no *controller* é executada.

#### 2.3.2.3.4 - afterRender

Chamada depois que uma ação é renderizada.

### 2.3.3 - A camada de visão

Essa camada é responsável de gerar a interface com o cliente. Na maioria das vezes o resultado dessa camada será um arquivo de extensão XHTML, mas é possível também gerar arquivos de RSS. É possível gerar também arquivos de extensão CSV.

Os arquivos da visão são salvos com a extensão .ctp e escritos em php. A extensão ctp quer dizer *CakePhp Template*. Esses arquivos possuem toda a lógica para receber os dados enviados pela camada de controle e apresentá-los para o usuário.

Os arquivos de visão são salvos no diretório app/views, dentro de uma pasta com o nome do controle que está relacionado. O nome do arquivo deve ser o nome da ação que a visão vai apresentar com extensão .ctp. Por exemplo, O arquivo de visão da ação de adicionar um produto do controle produto deve ser salvo no seguinte arquivo *app/views/produto/adicionar.ctp*

A camada de visão é composto de três partes distintas: *layout, elements* e *helpers*.

*Layouts* são arquivos de visão que irão abrigar diversos arquivos da aplicação. A maioria dos arquivos de visão são renderizadas dentro de um arquivo de *layout*.

*Elements* são pequenos pedaços de código que podem ser reusados em diversos lugares da aplicação. Geralmente são renderizados dentro de uma *view*.

*Helpers* fornecem lógica para os arquivos de visão. Através de *helpers* o CakePHP fornece formas fáceis de se construir formulários, aplicações Ajax, paginação, RSS, dentre diversas outras funcionalidades.

#### 2.3.3.1 - Layouts

O *layout* contém código de apresentação que abriga os arquivos de visão. Tudo que se deseja ver em uma visão deve ser posto dentro de um arquivo de *layout*.

Os arquivos de *layout* são postos dentro do diretório */app/views/layouts*. Por padrão o CakePHP irá procurar pelo arquivo */app/views/layouts/default.ctp* para renderizar os arquivos de visão dentro deste arquivo. Entretanto, é possível sobrescrever esse arquivo.

Um arquivo de *layout* deve conter uma região destinada para carregar os arquivos de visão. Isto é feito através da função *$content_for_layout*. Pode-se também definir o título da tela através de *$title_for_layout*. Para incorporar arquivos externos ao arquivo de *layout*, deve-se utilizar a função *$scripts_for_layout* e definir o nome dos arquivos que serão importados.

Ao invés de definir o título da visão em *layout* é possível definir no próprio arquivo de controle através de *$this->layout*. O mesmo ocorre com o título, que pode ser utilizado *$this->pageTitle* no arquivo de controle para definir o título da tela. Para ver um exemplo de arquivo de layout abra o arquivo */app/views/layouts/default.ctp* do CakePHP.

Exemplo de uso de *$this->pageTitle* e *$this->layout* no arquivo de controle

```
class UsuariosController extends AppController {
	function mostrarAlternativo() {
		$this->pageTitle = ':: Home ::';
		$this->layout = 'alternativo';
	}
	function mostrarPadrao() {
		$this->layout = 'padrao';
	}
}
```

Além do *layout* padrão que é o *default* o CakePHP também fornece o *ajax* que é adequado para renderizar telas com recursos *ajax* e o *layout* e o *flash* que é útil quando irá se lidar com mensagens enviadas via método *flash*.

#### 2.3.3.2 – Elements

Muitas aplicações possuem partes da apresentação que precisam ser repetidas em diversas partes. Para isso existe o *element*. Geralmente essas partes da apresentação podem ser um menu, formulários de login, caixas de ajuda, dentre diversos outros exemplos. Um elemento pode ser visto como um mini-arquivo de visão que é embutido dentro de um arquivo de visão, *layouts* e mesmo dentro de outro arquivo *element*.

Arquivos desse tipo são salvos dentro do diretório */app/views/elements/* e utilizam a extensão .ctp.

Exemplo de como importar um *Element*

```
	echo $this->element('helpbox');
```

#### 2.3.3.2.1 – Passando variáveis dentro de um elemento

É possível passar dados para um elemento através do segundo parâmetro.

```
$this->element('caixaDeMensagem',array("mensagem" => "nononoo nononoon"));
```

Neste exemplo, dentro de *element*, a variável $mensagem ficaria disponível com o valor "nononoo nononoon". Além de dados, a função *element* também fornece as opções *cache* e *plugin*.

#### 2.3.3.3 – Usando os Helpers básicos

O CakePHP carrega automaticamente os helpers Html e Form. Estes helpers são os mais utilizados no desenvolvimento de uma aplicação.

```
// Gerando uma imagem com o helper / HTML gerado
$html->image('nome_da_imagem.jpg',array('alt'=>'Texto alternativo para a imagem'));
<img src="/bookmarks/img/nome_da_imagem.jpg" alt="Texto alternativo para a imagem"/>

// Gerando um Link com o helper / HTML gerado
$html->link('texto do link','/bookmarks/add', array('class'=>'nomeDaClasseCss'));
<a href="/caminhoRelativo" class="nomeDaClasseCss">texto do link</a>

// Gerando um input com o helper de Form / HTML gerado
$form->input('Usuario.senha',array('type'=>'password'));
<input type="password" id="UsuarioSenha" value="" maxlength="32" name="data[Usuario]
[senha]">
```