



CakePHP

CakePHP Cookbook Documentation

Release 3.5

Cake Software Foundation

dez 03, 2017

Sumário

1	CakePHP num piscar de olhos	1
	Convenções Sobre Configuração	1
	A camada Model	1
	A camada View	2
	A camada Controller	2
	Ciclo de Requisições do CakePHP	3
	Apenas o Começo	4
	Leitura adicional	4
2	Guia de Início Rápido	11
	Tutorial - Criando um Bookmarker - Parte 1	11
	Tutorial - Criando um Bookmarker - Parte 2	17
3	3.0 - Guia de migração	25
	Requerimentos	25
	Ferramenta de atualização	25
	Layout do diretório da aplicação	25
	O CakePHP deve ser instalado via Composer	26
	Namespaces	26
	Constantes removidas	26
	Configuração	26
	Novo ORM	27
	Básico	27
	Debug	27
	Especificações/Configurações de objetos	27
	Cache	27
	Core	28
	Console	29
	Shell / Tarefa	29
	Eventos	30
	Log	30
	Roteamento	31
	Rede	33
	Sessões	33

Network\Http	34
Network>Email	34
Controller	34
Controller\Components	36
Model	38
Suíte de Testes	39
View	40
View\Helper	41
I18n	45
Localização	46
Testes	46
Utilitários	47
4 Tutoriais & Exemplos	51
Tutorial - Criando um Bookmarker - Parte 1	51
Tutorial - Criando um Bookmarker - Parte 2	57
Tutorial - Criando um Blog - Parte 1	63
Tutorial - Criando um Blog - Parte 2	67
Tutorial - Criando um Blog - Parte 3	77
Tutorial - Criando um Blog - Autenticação e Autorização	83
5 Contribuindo	91
Documentação	91
Tickets	99
Código	99
Padrões de codificação	102
Guia de retrocompatibilidade	112
6 Instalação	115
Requisitos	115
Instalando o CakePHP	116
Permissões	117
Servidor de Desenvolvimento	117
Produção	118
Aquecendo	118
Reescrita de URL	118
7 Configuração	123
Configurando sua Aplicação	123
Caminhos adicionais de classe	125
Configuração de Inflexão	126
Configurar classe	126
Lendo e escrevendo arquivos de configuração	128
Criando seus próprios mecanismos de configuração	130
Motores de Configuração Integrados	131
Bootstrapping CakePHP	132
Variáveis de Ambiente	133
Desabilitando tabelas genéricas	134
8 Roteamento	135
9 Routing	137
Quick Tour	137
Connecting Routes	139
Criando rotas RESTful	152

Passed Arguments	155
Generating URLs	156
Redirect Routing	157
Custom Route Classes	157
Creating Persistent URL Parameters	159
Handling Named Parameters in URLs	159
10 Objetos de requisição e resposta	161
Requisição	161
11 Controllers (Controladores)	163
O App Controller	163
Fluxo de requisições	164
Métodos (actions) de controllers	164
Redirecionando para outras páginas	167
Carregando models adicionais	168
Paginando um model	169
Configurando components para carregar	169
Configurando helpers para carregar	170
Ciclo de vida de callbacks em uma requisição	170
Mais sobre controllers	171
12 Views (Visualização)	175
A App View	175
View Templates	176
Usando View Blocks	179
Layouts	181
Elements	183
Eventos da View	186
Criando suas próprias Classes View	186
Mais sobre Views	187
13 Models (Modelos)	197
Exemplo rápido	197
Mais informação	199
14 Bake Console	281
Instalação	281
15 Caching	285
Configuring Cache Class	285
16 Console e Shells	287
O Console do CakePHP	287
Criando uma Shell	288
Tasks de Shell	290
Invocando outras Shells a partir da sua Shell	291
Recenendo Input de usuários	292
Criando Arquivos	292
Saída de dados do Console	292
Opções de configuração e Geração de ajuda	294
Roteamento em Shells / CLI	301
Métodos enganchados	302
Mais tópicos	302

17 Depuração	305
Depuração Básica	305
Usando a Classe Debugger	306
Valores de saída	306
Criando Logs com Pilha de Execução	306
Gerando Pilhas de Execução	306
Pegando Trechos de Arquivos	307
Usando Logging para Depuração	307
Debug Kit	308
18 Implantação	309
Atualizar config/app.php	309
Checar a segurança	310
Definir a raiz do documento	310
Aprimorar a performance de sua aplicação	310
19 Email	311
Uso Básico	311
Configuração	312
Definindo Cabeçalho	315
Enviando E-mail com Templates	315
Envio de Anexos	316
Usando Transportes	317
20 Erros & Exceções	319
Error & Exception Configuration	319
21 Sistema de eventos	321
22 Internacionalização e Localização	323
Configurando Traduções	323
Usando funções de tradução	325
Criar seus próprios Tradutores	329
23 Logging	331
Logging Configuration	331
24 Criando o Formulário	333
25 Processando Requisição de Dados	335
26 Definindo os Valores do Formulário	337
27 Pegando os Erros do Formulário	339
28 Invalidando Campos Individuais do Formulário no Controller	341
29 Criando o HTML com FormHelper	343
30 Plugins	345
Instalando um Plugin com Composer	345
Carregando um Plugin	346
Configuração do Plugin	347
Usando Plugins	348
Criando seus próprios complementos	348
Rotas para Plugin	349

Plugin Controllers	350
Plugin Models	351
Plugin Views	352
Plugin Assets	353
Components, Helpers and Behaviors	354
Expanda seu plugin	354
Publique seu plugin	355
31 REST	357
A Configuração é simples	357
Aceitando entrada em outros formatos	359
Roteamento RESTful	359
32 Segurança	361
Segurança	361
33 Sessions	365
Session Configuration	365
34 Testing	367
Instalando o PHPUnit	367
Configuração do banco de dados test	368
Running Tests	368
Fixtures	368
Controller Integration Testing	368
Testing Events	368
35 Validação	369
36 App Class	371
37 Collections (Coleções)	373
38 Arquivos & Pastas	375
39 Hash	377
40 Http Client	379
41 Inflector	381
Resumo dos métodos de Inflexão e Suas Saídas	381
Criando as formas singulares e plurais	382
Criando as formas CamelCase e nome_sublinhado	382
Criando formas legíveis para humanos	382
Criando formatos para nomes de tabelas e classes	383
Criando nomes de variáveis	383
Criando strings de URL seguras	383
Configuração da inflexão	383
42 Número	385
43 Objetos de Registro	387
44 Texto	389
45 Tempo	391

46 Xml	393
47 Constantes e Funções	395
Funções globais	395
Constantes de definição do Core	397
Constantes de definição de tempo	397
48 Debug Kit	399
Instalação	399
Armazenamento do DebugKit	399
Uso da barra de ferramentas	400
Usando o painel History	400
Desenvolvendo seus próprios painéis	401
49 Migrations	405
Instalação	405
Visão Geral	406
Criando migrations	407
Gerando migrações a partir de uma base de dados existente	411
Os Comandos	412
Usando migrations em plugins	415
Executando migrations em ambientes fora da linha de comando	415
Dicas e truques	417
50 Apêndices	421
Guia de Migração para a versão 3.x	421
Informações Gerais	423
PHP Namespace Index	427
Índice	429

CakePHP num piscar de olhos

O CakePHP é concebido para tornar tarefas de desenvolvimento web mais simples e fáceis. Por fornecer uma caixa de ferramentas completa para você poder começar, o CakePHP funciona bem em conjunto ou isoladamente.

O objetivo desta análise é introduzir os conceitos gerais presentes no CakePHP, e lhe dar uma rápida visão geral de como estes conceitos são implementados. Se você está ávido para começar um projeto, você pode [começar com o tutorial](#), ou mergulhar na documentação.

Convenções Sobre Configuração

O CakePHP provê uma estrutura organizacional básica que cobre nomenclaturas de classes, nomenclaturas de arquivos, nomenclaturas de banco de dados, e outras convenções. Apesar das convenções levarem algum tempo para serem assimiladas, ao segui-las o CakePHP evita configuração desnecessário e cria uma estrutura de aplicação uniforme que faz trabalhar com vários projetos uma tarefa suave. O [capítulo de convenções](#) cobre as variadas convenções que o CakePHP utiliza.

A camada Model

A camada Model representa a parte da sua aplicação que implementa a lógica de negócio. Ela é responsável por recuperar dados e convertê-los nos conceitos significativos primários na sua aplicação. Isto inclui processar, validar, associar ou qualquer outra tarefa relacionada à manipulação de dados.

No caso de uma rede social, a camada Model deveria tomar cuidado de tarefas como salvar os dados do usuário, salvar as associações entre amigos, salvar e recuperar fotos de usuários, localizar sugestões para novos amigos, etc. Os objetos de modelo podem ser pensados como “Friend”, “User”, “Comment”, ou “Photo”. Se nós quiséssemos carregar alguns dados da nossa tabela `users` poderíamos fazer:

```
use Cake\ORM\TableRegistry;

$users = TableRegistry::get('Users');
$query = $users->find();
foreach ($query as $row) {
    echo $row->username;
}
```

Você pode notar que não precisamos escrever nenhum código antes de podermos começar a trabalhar com nossos dados. Por usar convenções, o CakePHP irá utilizar classes padrão para tabelas e entidades ainda não definidas.

Se nós quiséssemos criar um usuário e salvá-lo (com validação) fariamos algo assim:

```
use Cake\ORM\TableRegistry;

$users = TableRegistry::get('Users');
$user = $users->newEntity(['email' => 'mark@example.com']);
$users->save($user);
```

A camada View

A View renderiza uma apresentação de dados modelados. Estando separada dos objetos da Model, é responsável por utilizar a informação que tem disponível para produzir qualquer interface de apresentação que a sua aplicação possa precisar.

Por exemplo, a view pode usar dados da model para renderizar uma página HTML que os contenha, ou um resultado formatado como XML:

```
// No arquivo view, nós renderizaremos um 'elemento' para cada usuário.
<?php foreach ($users as $user): ?>
    <div class="user">
        <?= $this->element('user', ['user' => $user]) ?>
    </div>
<?php endforeach; ?>
```

A camada View provê alguma variedade de extensões como *Elements* e *View Cells (Células de Visualização)* para permitir que você reutilize sua lógica de apresentação.

A camada View não está limitada somente a HTML ou apresentação textual dos dados. Ela pode ser usada para entregar formatos de dado comuns como JSON, XML, e através de uma arquitetura encaixável qualquer outro formato que você venha precisar.

A camada Controller

A camada Controller manipula requisições dos usuários. É responsável por renderizar uma resposta com o auxílio de ambas as camadas, Model e View respectivamente.

Um controller pode ser visto como um gerente que certifica-se que todos os recursos necessários para completar uma tarefa sejam delegados aos trabalhadores corretos. Ele aguarda por petições dos clientes, checa suas validades de acordo com autenticação ou regras de autorização, delega requisições ou processamento de dados da camada Model, selecciona o tipo de dados de apresentação que os clientes estão aceitando, e finalmente delega o processo de renderização para a camada View. Um exemplo de controller para registro de usuário seria:

```
public function add()
{
    $user = $this->Users->newEntity();
    if ($this->request->is('post')) {
        $user = $this->Users->patchEntity($user, $this->request->getData());
        if ($this->Users->save($user, ['validate' => 'registration'])) {
            $this->Flash->success(__('Você está registrado.));
        } else {
            $this->Flash->error(__('Houve algum problema.));
        }
    }
}
```

```

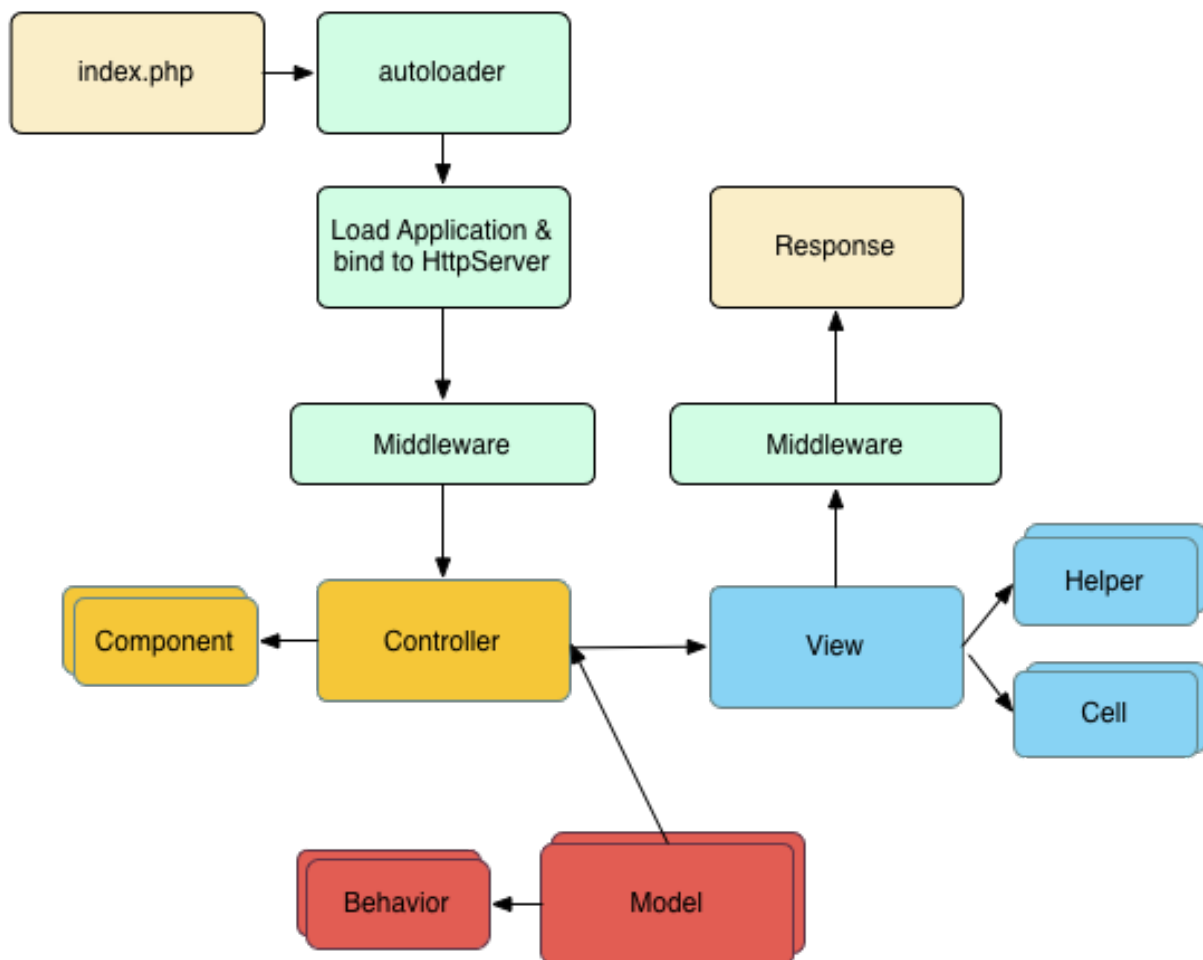
    }
}
$this->set('user', $user);
}

```

Você pode perceber que nós nunca renderizamos uma view explicitamente. As convenções do CakePHP tomarão cuidado de selecionar a view correta e renderizá-la como os dados definidos com `set()`.

Ciclo de Requisições do CakePHP

Agora que você é familiar com as diferentes camadas no CakePHP, vamos revisar como um ciclo de requisição funciona no CakePHP:



O ciclo de requisição do CakePHP começa com a solicitação de uma página ou recurso da sua aplicação, seguindo a cadência abaixo:

1. As regras de reescrita do servidor encaminham a requisição para **webroot/index.php**.
2. Sua aplicação é carregada e vinculada a um `HttpServer`.
3. O *middleware* da sua aplicação é inicializado.

4. A requisição e a resposta são processados através do *PSR-7 Middleware* que sua aplicação utiliza. Normalmente isso inclui captura de erros e roteamento.
5. Se nenhuma resposta for retornada do *middleware* e a requisição contiver informações de rota, um *Controller* e uma *action* são acionados.
6. A *action* do *Controller* é chamada e o mesmo interage com os *Models* e *Components* requisitados.
7. O *controller* delega a responsabilidade de criar respostas à *view*, para assim gerar a saída de dados resultante do *Model*.
8. A *View* utiliza *Helpers* e *Cells* para gerar o corpo e cabeçalho das respostas.
9. A resposta é enviada de volta através do `/controllers/middleware`.
10. O `HttpServer` emite a resposta para o servidor web.

Apenas o Começo

Esperamos que essa rápida visão geral tenha despertado seu interesse. Alguns outros grandes recursos no CakePHP são:

- *Framework de cache* que integra com Memcached, Redis e outros backends.
- Poderosas *ferramentas de geração de código* para você sair em disparada.
- *Framework de teste integrado* para você assegurar-se que seu código funciona perfeitamente.

Os próximos passos óbvios são *baixar o CakePHP*, ler o *tutorial e construir algo fantástico*.

Leitura adicional

Onde Conseguir Ajuda

O website oficial do CakePHP

<https://cakephp.org>

O website oficial do CakePHP é sempre um ótimo lugar para visitar. Ele provê links para ferramentas comumente utilizadas por desenvolvedores, screencasts, oportunidades de doação e downloads.

O Cookbook

<https://book.cakephp.org>

Esse manual deveria ser o primeiro lugar para onde você iria afim de conseguir respostas. Assim como muitos outros projetos de código aberto, nós conseguimos novos colaboradores regularmente. Tente o seu melhor para responder suas questões por si só. Respostas vão vir lentamente, e provavelmente continuarão longas. Você pode suavizar nossa carga de suporte. Tanto o manual quanto a API possuem um componente online.

A Bakery

<https://bakery.cakephp.org>

A “padaria” do CakePHP é um local para todas as coisas relacionadas ao CakePHP. Visite-a para tutoriais, estudos de caso e exemplos de código. Uma vez que você tenha se familiarizado com o CakePHP, autentique-se e compartilhe seu conhecimento com a comunidade, ganhe instantaneamente fama e fortuna.

A API

<https://api.cakephp.org/>

Diretamente ao ponto, dos desenvolvedores do núcleo do CakePHP, a API (Application Programming Interface) do CakePHP é a mais compreensiva documentação sobre os detalhes técnicos e minuciosos sobre do funcionamento interno do framework.

Os Testes de Caso

Se você sente que a informação provida pela API não é suficiente, verifique os códigos de testes de caso do CakePHP. Eles podem servir como exemplos práticos para funções e e utilização de dados referentes a uma classe.:

```
tests/TestCase/
```

O canal de IRC

Canal de IRC na irc.freenode.net:

- #cakephp – Discussão geral
- #cakephp-docs – Documentação
- #cakephp-bakery – Bakery
- #cakephp-fr – Canal francês.

Se você está travado, nos faça uma visita no canal de IRC do CakePHP. Alguém do [time de desenvolvimento](#)⁴ normalmente está conectado, especialmente nos horários diurnos da América do Sul e América do Norte. Nós apreciaríamos ouvi-lo se você precisar de ajuda, se quiser encontrar usuários da sua área ou ainda se quiser doar seu novo carro esporte.

Stackoverflow

<https://stackoverflow.com/>⁵

Marque suas questões com a tag `cakephp` e especifique a versão que você está utilizando para permitir que usuários do stackoverflow achem suas questões.

⁴ <https://github.com/cakephp?tab=members>

⁵ <https://stackoverflow.com/questions/tagged/cakephp/>

Onde conseguir ajuda em sua língua

Francês

- [Comunidade CakePHP francesa](#)⁶

Português brasileiro

- [Comunidade CakePHP brasileira](#)⁷

Convenções do CakePHP

Nós somos grandes fãs de convenção sobre configuração. Apesar de levar um pouco de tempo para aprender as convenções do CakePHP, você economiza tempo a longo prazo. Ao seguir as convenções, você ganha funcionalidades instantaneamente e liberta-se do pesadelo de manutenção e rastreamento de arquivos de configuração. Convenções também prezam por uma experiência de desenvolvimento uniforme, permitindo que outros desenvolvedores ajudem mais facilmente.

Convenções para Controllers

Os nomes das classes de Controllers são pluralizados, CamelCased, e terminam em `Controller`. `PeopleController` e `LatestArticlesController` são exemplos de nomes convencionais para controllers.

Métodos públicos nos Controllers são frequentemente referenciados como ‘actions’ acessíveis através de um navegador web. Por exemplo, o `/articles/view` mapeia para o método `view()` do `ArticlesController` sem nenhum esforço. Métodos privados ou protegidos não podem ser acessados pelo roteamento.

Considerações de URL para nomes de Controller

Como você acabou de ver, controllers singulares mapeiam facilmente um caminho simples, todo em minúsculo. Por exemplo, `ApplesController` (o qual deveria ser definido no arquivo de nome ‘`ApplesController.php`’) é acessado por <http://example.com/apples>.

Controllers com múltiplas palavras *podem* estar em qualquer forma ‘flexionada’ igual ao nome do controller, então:

- `/redApples`
- `/RedApples`
- `/Red_apples`
- `/red_apples`

Todos resolverão para o index do controller `RedApples`. Porém, a forma correta é que suas URLs sejam minúsculas e separadas por sublinhado, portanto `/red_apples/go_pick` é a forma correta de acessar a action `RedApplesController::go_pick`.

Quando você cria links usando `this->Html->link()`, você pode usar as seguintes convenções para a array de url:

⁶ <http://cakephp-fr.org>

⁷ <http://cakephp-br.org>

```
$this->Html->link('link-title', [
    'prefix' => 'MyPrefix' // CamelCased
    'plugin' => 'MyPlugin', // CamelCased
    'controller' => 'ControllerName', // CamelCased
    'action' => 'actionName' // camelBacked
])
```

Para mais informações sobre o manuseio de URLs e parâmetros do CakePHP, veja [Connecting Routes](#).

Convenções para nomes de Classes e seus nomes de arquivos

No geral, nomes de arquivos correspondem aos nomes das classes, e seguem os padrões PSR-0 ou PSR-4 para auto-carregamento. A seguir seguem exemplos de nomes de classes e de seus arquivos:

- A classe de Controller **KissesAndHugsController** deveria ser encontrada em um arquivo nomeado **KissesAndHugsController.php**
- A classe de Component **MyHandyComponent** deveria ser encontrada em um arquivo nomeado **MyHandyComponent.php**
- A classe de Table **OptionValuesTable** deveria ser encontrada em um arquivo nomeado **OptionValuesTable.php**.
- A classe de Entity **OptionValue** deveria ser encontrada em um arquivo nomeado **OptionValue.php**.
- A classe de Behavior **EspecialyFunkableBehavior** deveria ser encontrada em um arquivo nomeado **EspecialyFunkableBehavior.php**
- A classe de View **SuperSimpleView** deveria ser encontrada em um arquivo nomeado **SuperSimpleView.php**
- A classe de Helper **BestEverHelper** deveria ser encontrada em um arquivo nomeado **BestEverHelper.php**

Cada arquivo deveria estar localizado no diretório/namespace apropriado de sua aplicação.

Convenções para Models e Databases

Os nomes de classe de Tables são pluralizadas e CamelCased. People, BigPeople, and ReallyBigPeople são todos exemplos convencionais de models.

Os nomes de Tables correspondentes aos models do CakePHP são pluralizadas e separadas por sublinhado. As tables sublinhadas para os models mencionados acima seriam people, big_people, e really_big_people, respectively.

Você pode utilizar a biblioteca utility `Cake\Utility\Inflector` para checar o singular/plural de palavras. Veja o [Inflector](#) para mais informações. Recomenda-se que as tables sejam criadas e mantidas na língua inglesa.

Campos com duas ou mais palavras são separados por sublinhado: first_name.

Chaves estrangeiras nos relacionamentos hasMany, belongsTo ouhasOne são reconhecidas por padrão como o nome (singular) da table relacionada seguida por _id. Então se Bakers hasMany Cakes, a table cakes irá referenciar-se para a table bakers através da chave estrangeira baker_id. Para uma tabela como category_types a qual o nome contém mais palavras, a chave estrangeira seria a category_type_id.

tables de união, usadas no relacionamento BelongsToMany entre models, devem ser nomeadas depois das tables que ela está unindo, ordenadas em ordem alfabética (apples_zebras ao invés de zebras_apples).

Convenções para Views

Arquivos de template views são nomeadas seguindo as funções que a exibem do controller, separadas por sublinhado. A função `getReady()` da classe `PeopleController` buscará por um template view em `src/Template/People/get_ready.ctp`. O padrão é `src/Template/Controller/underscored_function_name.ctp`.

Por nomear as partes de sua aplicação utilizando as convenções do CakePHP, você ganha funcionalidades sem luta e sem amarras de configuração. Aqui está um exemplo final que enlaça as convenções juntas:

- Table: “people”
- Classe Table: “PeopleTable”, encontrada em `src/Model/Table/PeopleTable.php`
- Classe Entity: “Person”, encontrada em `src/Model/Entity/Person.php`
- Classe Controller: “PeopleController”, encontrada em `src/Controller/PeopleController.php`
- View template, encontrado em `src/Template/People/index.ctp`

Utilizando estas convenções, o CakePHP sabe que uma requisição para <http://example.com/people/> mapeia para uma chamada da função `index()` do `PeopleController`, onde o model `Person` é automaticamente disponibilizado (e automaticamente amarrado à table ‘people’ no banco de dados), e então renderiza-se um arquivo view template. Nenhuma destes relacionamentos foi configurado de qualquer forma se não por criar classes e arquivos que você precisaria criar de qualquer forma.

Agora que você foi introduzido aos fundamentos do CakePHP, você pode tentar seguir através do [Tutorial - Criando um Blog - Parte 1](#) para ver como as coisas se encaixam juntas.

Estrutura de pastas do CakePHP

Depois de você ter baixado e extraído o CakePHP, aí estão os arquivos e pastas que você deve ver:

- bin
- config
- logs
- plugins
- src
- tests
- tmp
- vendor
- webroot
- .htaccess
- composer.json
- index.php
- README.md

Você notará alguns diretórios principais:

- O diretório *bin* contem os executáveis por console do Cake.
- O diretório *config* contem os (poucos) *Configuração* arquivos de configuração que o CakePHP utiliza. Detalhes de conexão com banco de dados, inicialização, arquivos de configuração do núcleo da aplicação, e relacionados devem ser postos aqui.

- O diretório *logs* será normalmente onde seus arquivos de log ficarão, dependendo das suas configurações.
- O diretório *plugins* será onde *Plugins* que sua aplicação utiliza serão armazenados.
- O diretório *src* será onde você fará sua mágica: é onde os arquivos da sua aplicação serão colocados.
- O diretório *tests* será onde você colocará os casos de teste para sua aplicação.
- O diretório *tmp* será onde o CakePHP armazenará dados temporários. O modo como os dados serão armazenados depende da configuração do CakePHP, mas esse diretório é comumente usado para armazenar descrições de modelos e algumas vezes informação de sessão.
- O diretório *vendor* será onde o CakePHP e outras dependências da aplicação serão instalados. Faça uma nota pessoal para **não** editar arquivos deste diretório. Nós não podemos ajudar se você tiver-lo feito.
- O diretório *webroot* será a raiz pública de documentos da sua aplicação. Ele contém todos os arquivos que você gostaria que fossem públicos.

Certifique-se que os diretórios *tmp* e *logs* existem e são passíveis de escrita, senão a performance de sua aplicação será severamente impactada. Em modo de debug, o CakePHP irá alertá-lo se este for o caso.

O diretório *src*

O diretório *src* do CakePHP é onde você fará a maior parte do desenvolvimento de sua aplicação. Vamos ver mais de perto a estrutura de pastas dentro de *src*.

Console Contém os comandos e tarefas de console para sua aplicação. Para mais informações veja *Console e Shells*.

Controller Contém os controllers de sua aplicação e seus componentes.

Locale Armazena arquivos textuais para internacionalização.

Model Contém as tables, entities e behaviors de sua aplicação.

View Classes de apresentação são alocadas aqui: cells, helpers, e arquivos view.

Template Arquivos de apresentação são alocados aqui: elements, páginas de erro, layouts, e templates view.

Guia de Início Rápido

A melhor forma de viver experiências e aprender sobre CakePHP é sentar e construir algo. Para começar nós iremos construir uma aplicação simples de blog.

Tutorial - Criando um Bookmarker - Parte 1

Esse tutorial vai guiar você através da criação de uma simples aplicação de marcação (bookmarker). Para começar, nós vamos instalar o CakePHP, criar nosso banco de dados, e usar as ferramentas que o CakePHP fornece para obter nossa aplicação de pé rápido.

Aqui está o que você vai precisar:

1. Um servidor de banco de dados. Nós vamos usar o servidor MySQL neste tutorial. Você precisa saber o suficiente sobre SQL para criar um banco de dados: O CakePHP vai tomar as rédeas a partir daí. Por nós estarmos usando o MySQL, também certifique-se que você tem a extensão `pdo_mysql` habilitada no PHP.
2. Conhecimento básico sobre PHP.

Vamos começar!

Instalação do CakePHP

A maneira mais fácil de instalar o CakePHP é usando Composer, um gerenciador de dependências para o PHP. É uma forma simples de instalar o CakePHP a partir de seu terminal ou prompt de comando. Primeiro, você precisa baixar e instalar o Composer. Se você tiver instalada a extensão cURL do PHP, execute o seguinte comando:

```
curl -s https://getcomposer.org/installer | php
```

Ao invés disso, você também pode baixar o arquivo `composer.phar` do [site](https://getcomposer.org/)⁸ oficial.

Em seguida, basta digitar a seguinte linha no seu terminal a partir do diretório onde se localiza o arquivo `composer.phar` para instalar o esqueleto de aplicações do CakePHP no diretório `bookmarker`.

```
php composer.phar create-project --prefer-dist cakephp/app bookmarker
```

⁸ <https://getcomposer.org/download/>

A vantagem de usar Composer é que ele irá completar automaticamente um conjunto importante de tarefas, como configurar as permissões de arquivo e criar a sua **config/app.php**.

Há outras maneiras de instalar o CakePHP. Se você não puder ou não quiser usar Composer, veja a seção [Instalação](#).

Independentemente de como você baixou o CakePHP, uma vez que sua instalação for concluída, a estrutura dos diretórios deve ficar parecida com o seguinte:

```
/bookmarker
  /bin
  /config
  /logs
  /plugins
  /src
  /tests
  /tmp
  /vendor
  /webroot
  .editorconfig
  .gitignore
  .htaccess
  .travis.yml
  composer.json
  index.php
  phpunit.xml.dist
  README.md
```

Agora pode ser um bom momento para aprender sobre como a estrutura de diretórios do CakePHP funciona: Confira a seção [Estrutura de pastas do CakePHP](#).

Verificando nossa instalação

Podemos checar rapidamente que a nossa instalação está correta, verificando a página inicial padrão. Antes que você possa fazer isso, você vai precisar iniciar o servidor de desenvolvimento:

```
bin/cake server
```

Isto irá iniciar o servidor embutido do PHP na porta 8765. Abra <http://localhost:8765> em seu navegador para ver a página de boas-vindas. Todas as verificações devem estar checadadas corretamente, a não ser a conexão com banco de dados do CakePHP. Se não, você pode precisar instalar extensões do PHP adicionais, ou definir permissões de diretório.

Criando o banco de dados

Em seguida, vamos criar o banco de dados para a nossa aplicação. Se você ainda não tiver feito isso, crie um banco de dados vazio para uso nesse tutorial, com um nome de sua escolha, por exemplo, `cake_bookmarks`. Você pode executar o seguinte SQL para criar as tabelas necessárias:

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  email VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  created DATETIME,
  modified DATETIME
);
```

```

CREATE TABLE bookmarks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(50),
    description TEXT,
    url TEXT,
    created DATETIME,
    modified DATETIME,
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
);

CREATE TABLE bookmarks_tags (
    bookmark_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (bookmark_id, tag_id),
    INDEX tag_idx (tag_id, bookmark_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks(id)
);

```

Você deve ter notado que a tabela `bookmarks_tags` utilizada uma chave primária composta. O CakePHP suporta chaves primárias compostas quase todos os lugares, tornando mais fácil construir aplicações multi-arrendados.

Os nomes de tabelas e colunas que usamos não foram arbitrários. Usando *convenções de nomenclatura* do CakePHP, podemos alavancar o desenvolvimento e evitar ter de configurar o framework. O CakePHP é flexível o suficiente para acomodar até mesmo esquemas de banco de dados legados inconsistentes, mas aderir às convenções vai lhe poupar tempo.

Configurando o banco de dados

Em seguida, vamos dizer ao CakePHP onde o nosso banco de dados está como se conectar a ele. Para muitos, esta será a primeira e última vez que você vai precisar configurar qualquer coisa.

A configuração é bem simples: basta alterar os valores do array `Datasources.default` no arquivo **config/app.php** pelos que se aplicam à sua configuração. A amostra completa da gama de configurações pode ser algo como o seguinte:

```

return [
    // Mais configuração acima.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_bookmarks',
            'encoding' => 'utf8',

```

```
        'timezone' => 'UTC',
        'cacheMetadata' => true,
    ],
    ],
    // Mais configuração abaixo.
];
```

Depois de salvar o seu arquivo **config/app.php**, você deve notar que a mensagem ‘CakePHP is able to connect to the database’ tem uma marca de verificação.

Nota: Uma cópia do arquivo de configuração padrão do CakePHP é encontrado em **config/app.default.php**.

Gerando o código base

Devido a nosso banco de dados seguir as convenções do CakePHP, podemos usar o *bake console* para gerar rapidamente uma aplicação básica. Em sua linha de comando execute:

```
bin/cake bake all users
bin/cake bake all bookmarks
bin/cake bake all tags
```

Isso irá gerar os controllers, models, views, seus casos de teste correspondentes, e fixtures para os nossos users, bookmarks e tags. Se você parou seu servidor, reinicie-o e vá para `http://localhost:8765/bookmarks`.

Você deverá ver uma aplicação que dá acesso básico, mas funcional a tabelas de banco de dados. Adicione alguns users, bookmarks e tags.

Adicionando criptografia de senha

Quando você criou seus users, você deve ter notado que as senhas foram armazenadas como texto simples. Isso é muito ruim do ponto de vista da segurança, por isso vamos consertar isso.

Este também é um bom momento para falar sobre a camada de modelo. No CakePHP, separamos os métodos que operam em uma coleção de objetos, e um único objeto em diferentes classes. Métodos que operam na recolha de entidades são colocadas na classe *Table*, enquanto as características pertencentes a um único registro são colocados na classe *Entity*.

Por exemplo, a criptografia de senha é feita no registro individual, por isso vamos implementar esse comportamento no objeto entidade. Dada a circunstância de nós querermos criptografar a senha cada vez que é definida, vamos usar um método modificador/definidor. O CakePHP vai chamar métodos de definição baseados em convenções a qualquer momento que uma propriedade é definida em uma de suas entidades. Vamos adicionar um definidor para a senha. Em **src/Model/Entity/User.php** adicione o seguinte:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Auth\DefaultPasswordHasher;

class User extends Entity
{
    // Code from bake.
```

```
protected function _setPassword($value)
{
    $hasher = new DefaultPasswordHasher();
    return $hasher->hash($value);
}
```

Agora atualize um dos usuários que você criou anteriormente, se você alterar sua senha, você deve ver um senha criptografada ao invés do valor original nas páginas de lista ou visualização. O CakePHP criptografa senhas com `bcrypt`⁹ por padrão. Você também pode usar `sha1` ou `md5` caso venha a trabalhar com um banco de dados existente.

Recuperando bookmarks com uma tag específica

Agora que estamos armazenando senhas com segurança, podemos construir algumas características mais interessantes em nossa aplicação. Uma vez que você acumulou uma coleção de bookmarks, é útil ser capaz de pesquisar através deles por tag. Em seguida, vamos implementar uma rota, a ação do controller, e um método localizador para pesquisar através de bookmarks por tag.

Idealmente, nós teríamos uma URL que se parece com `http://localhost:8765/bookmarks/tagged/funny/cat/gifs`. Isso deveria nos permitir a encontrar todos os bookmarks que têm as tags ‘funny’, ‘cat’ e ‘gifs’. Antes de podermos implementar isso, vamos adicionar uma nova rota. Em `config/routes.php`, adicione o seguinte na parte superior do arquivo:

```
Router::scope(
    '/bookmarks',
    ['controller' => 'Bookmarks'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);
```

O acima define uma nova “rota” que liga o caminho `/bookmarks/tagged/*`, a `BookmarksController::tags()`. Ao definir rotas, você pode isolar como suas URLs parecerão, de como eles são implementadas. Se fôssemos visitar `http://localhost:8765/bookmarks/tagged`, veríamos ver uma página de erro informativa do CakePHP. Vamos implementar esse método ausente agora. Em `src/Controller/BookmarksController.php` adicione o seguinte:

```
public function tags()
{
    $tags = $this->request->getParam('pass');
    $bookmarks = $this->Bookmarks->find('tagged', [
        'tags' => $tags
    ]);
    $this->set(compact('bookmarks', 'tags'));
}
```

Criando o método localizador

No CakePHP nós gostamos de manter as nossas ações do controller enxutas, e colocar a maior parte da lógica de nossa aplicação nos modelos. Se você fosse visitar a URL `/bookmarks/tagged` agora, você veria um erro sobre o método `findTagged` não estar implementado ainda, então vamos fazer isso. Em `src/Model/Table/BookmarksTable.php` adicione o seguinte:

⁹ <http://codahale.com/how-to-safely-store-a-password/>

```
public function findTagged(Query $query, array $options)
{
    $bookmarks = $this->find()
        ->select(['id', 'url', 'title', 'description']);

    if (empty($options['tags'])) {
        $bookmarks
            ->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        $bookmarks
            ->innerJoinWith('Tags')
            ->where(['Tags.title IN ' => $options['tags']]);
    }

    return $bookmarks->group(['Bookmarks.id']);
}
```

Nós implementamos um método *localizador customizado*. Este é um conceito muito poderoso no CakePHP que lhe permite construir consultas reutilizáveis. Em nossa pesquisa, nós alavancamos o método `matching()` que nos habilita encontrar bookmarks que têm uma tag ‘correspondente’.

Criando a view

Agora, se você visitar a URL `/bookmarks/tagged`, o CakePHP irá mostrar um erro e deixá-lo saber que você ainda não fez um arquivo view. Em seguida, vamos construir o arquivo view para a nossa ação `tags`. Em `src/Template/Bookmarks/tags.ctp` coloque o seguinte conteúdo:

```
<h1>
    Bookmarks tagged with
    <?= $this->Text->toList(h($tags)) ?>
</h1>

<section>
<?php foreach ($bookmarks as $bookmark): ?>
    <article>
        <h4><?= $this->Html->link($bookmark->title, $bookmark->url) ?></h4>
        <small><?= h($bookmark->url) ?></small>
        <?= $this->Text->autoParagraph(h($bookmark->description)) ?>
    </article>
<?php endforeach; ?>
</section>
```

O CakePHP espera que os nossos templates sigam a convenção de nomenclatura onde o nome do template é a versão minúscula e grifada do nome da ação do controller.

Você pode perceber que fomos capazes de utilizar as variáveis `$tags` e `bookmarks` em nossa view. Quando usamos o método `set()` em nosso controller, automaticamente definimos variáveis específicas que devem ser enviadas para a view. A view vai tornar todas as variáveis passadas disponíveis nos templates como variáveis locais.

Em nossa view, usamos alguns dos *helpers* nativos do CakePHP. Helpers são usados para criar lógica re-utilizável para a formatação de dados, a criação de HTML ou outra saída da view.

Agora você deve ser capaz de visitar a URL `/bookmarks/tagged/funny` e ver todas os bookmarks com a tag ‘funny’.

Até agora, nós criamos uma aplicação básica para gerenciar bookmarks, tags e users. No entanto, todos podem ver as tags de toda a gente. No próximo capítulo, vamos implementar a autenticação e restringir os bookmarks visíveis para somente aqueles que pertencem ao usuário atual.

Agora vá a [Tutorial - Criando um Bookmarker - Parte 2](#) para continuar a construir sua aplicação ou mergulhe na documentação para saber mais sobre o que CakePHP pode fazer por você.

Tutorial - Criando um Bookmarker - Parte 2

Depois de terminar a *primeira parte deste tutorial*, você deve ter uma aplicação muito básica. Neste capítulo iremos adicionar autenticação e restringir as bookmarks para que cada usuário possa ver/modificar somente aquelas que possam.

Adicionando login

No CakePHP, a autenticação é feita por *Components (Componentes)*. Os Components podem ser considerados como formas de criar pedaços reutilizáveis de código relacionado a controllers com uma característica específica ou conceito. Os components também podem ligar-se ao evento do ciclo de vida do controller e interagir com a sua aplicação. Para começar, vamos adicionar o AuthComponent a nossa aplicação. Nós vamos querer muito que cada método exija autenticação, por isso vamos acrescentar o *AuthComponent* em nosso ApplicationController:

```
// Em src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ]
        ]);

        // Permite a ação display, assim nosso pages controller
        // continua a funcionar.
        $this->Auth->allow(['display']);
    }
}
```

Acabamos de dizer ao CakePHP que queremos carregar os components Flash e Auth. Além disso, temos a configuração personalizada do AuthComponent, assim a nossa tabela users pode usar email como username. Agora, se

you for a qualquer URL, você vai ser chutado para `/users/login`, que irá mostrar uma página de erro já que não escrevemos o código ainda. Então, vamos criar a ação de login:

```
// Em src/Controller/UsersController.php

public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Your username or password is incorrect.');
```

E em `src/Template/Users/login.ctp` adicione o seguinte:

```
<h1>Login</h1>
<?= $this->Form->create() ?>
<?= $this->Form->input('email') ?>
<?= $this->Form->input('password') ?>
<?= $this->Form->button('Login') ?>
<?= $this->Form->end() ?>
```

Agora que temos um simples formulário de login, devemos ser capazes de efetuar login com um dos users que tenham senha criptografada.

Nota: Se nenhum de seus users tem senha criptografada, comente a linha `loadComponent('Auth')`. Então vá e edite o user, salvando uma nova senha para ele.

Agora você deve ser capaz de entrar. Se não, certifique-se que você está usando um user que tenha senha criptografada.

Adicionando logout

Agora que as pessoas podem efetuar o login, você provavelmente vai querer fornecer uma maneira de encerrar a sessão também. Mais uma vez, no `UsersController`, adicione o seguinte código:

```
public function logout()
{
    $this->Flash->success('You are now logged out.');
```

Agora você pode visitar `/users/logout` para sair e ser enviado à página de login.

Ativando inscrições

Se você não estiver logado e tentar visitar `/usuários` / adicionar você vai ser expulso para a página de login. Devemos corrigir isso se quisermos que as pessoas se inscrevam em nossa aplicação. No `UsersController` adicione o seguinte:

```
public function beforeFilter(\Cake\Event\Event $event)
{
```

```
$this->Auth->allow(['add']);
}
```

O texto acima diz ao AuthComponent que a ação add não requer autenticação ou autorização. Você pode querer dedicar algum tempo para limpar a /users/add e remover os links enganosos, ou continuar para a próxima seção. Nós não estaremos construindo a edição do usuário, visualização ou listagem neste tutorial, então eles não vão funcionar, já que o AuthComponent vai negar-lhe acesso a essas ações do controller.

Restringindo acesso

Agora que os usuários podem conectar-se, nós vamos querer limitar os bookmarks que podem ver para aqueles que fizeram. Nós vamos fazer isso usando um adaptador de 'autorização'. Sendo os nossos requisitos bastante simples, podemos escrever um código em nossa BookmarksController. Mas antes de fazer isso, vamos querer dizer ao AuthComponent como nossa aplicação vai autorizar ações. Em seu ApplicationController adicione o seguinte:

```
public function isAuthorized($user)
{
    return false;
}
```

Além disso, adicione o seguinte à configuração para Auth em seu ApplicationController:

```
'authorize' => 'Controller',
```

Seu método initialize agora deve parecer com:

```
public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize' => 'Controller', //added this line
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Permite a ação display, assim nosso pages controller
    // continua a funcionar.
    $this->Auth->allow(['display']);
}
```

Vamos usar como padrão, negação do acesso, e de forma incremental conceder acesso onde faça sentido. Primeiro, vamos adicionar a lógica de autorização para os bookmarks. Em seu BookmarksController adicione o seguinte:

```
public function isAuthorized($user)
{
```

```

$action = $this->request->params['action'];

// As ações add e index são permitidas sempre.
if (in_array($action, ['index', 'add', 'tags'])) {
    return true;
}
// Todas as outras ações requerem um id.
if (!$this->request->getParam('pass.0')) {
    return false;
}

// Checa se o bookmark pertence ao user atual.
$id = $this->request->getParam('pass.0');
$bookmark = $this->Bookmarks->get($id);
if ($bookmark->user_id == $user['id']) {
    return true;
}
return parent::isAuthorized($user);
}

```

Agora, se você tentar visualizar, editar ou excluir um bookmark que não pertença a você, você deve ser redirecionado para a página de onde veio. No entanto, não há nenhuma mensagem de erro sendo exibida, então vamos corrigir isso a seguir:

```

// In src/Template/Layout/default.ctp
// Under the existing flash message.
<?= $this->Flash->render('auth') ?>

```

Agora você deve ver as mensagens de erro de autorização.

Corrigindo a view de listagem e formulários

Enquanto view e delete estão trabalhando, edit, add e index tem alguns problemas:

1. Ao adicionar um bookmark, você pode escolher o user.
2. Ao editar um bookmark, você pode escolher o user.
3. A página de listagem mostra os bookmarks de outros users.

Vamos enfrentar o formulário de adição em primeiro lugar. Para começar remova o input ('user_id') a partir de `src/Template/Bookmarks/add.ctp`. Com isso removido, nós também vamos atualizar o método add:

```

public function add()
{
    $bookmark = $this->Bookmarks->newEntity();
    if ($this->request->is('post')) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
        ↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
```

```
$this->set(compact('bookmark', 'tags'));
}
```

Ao definir a propriedade da entidade com os dados da sessão, nós removemos qualquer possibilidade do user modificar de que outro user um bookmark seja. Nós vamos fazer o mesmo para o formulário edit e action edit. Sua ação edit deve ficar assim:

```
public function edit($id = null)
{
    $bookmark = $this->Bookmarks->get($id, [
        'contain' => ['Tags']
    ]);
    if ($this->request->is(['patch', 'post', 'put'])) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
        ↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
```

```
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
```

```
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
}
```

View de listagem

Agora, nós precisamos apenas exibir bookmarks para o user logado. Nós podemos fazer isso ao atualizar a chamada para `paginate()`. Altere sua ação index:

```
public function index()
{
    $this->paginate = [
        'conditions' => [
            'Bookmarks.user_id' => $this->Auth->user('id'),
        ]
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
}
```

Nós também devemos atualizar a ação `tags()` e o método localizador relacionado, mas vamos deixar isso como um exercício para que você conclua por si.

Melhorando a experiência com as tags

Agora, adicionar novas tags é um processo difícil, pois o `TagsController` proíbe todos os acessos. Em vez de permitir o acesso, podemos melhorar a interface do usuário para selecionar tags usando um campo de texto separado por vírgulas. Isso permitirá dar uma melhor experiência para os nossos usuários, e usar mais alguns grandes recursos no ORM.

Adicionando um campo computado

Porque nós queremos uma maneira simples de acessar as tags formatados para uma entidade, podemos adicionar um campo virtual/computado para a entidade. Em **src/Model/Entity/Bookmark.php** adicione o seguinte:

```
use Cake\Collection\Collection;

protected function _getTagString()
{
    if (isset($this->_properties['tag_string'])) {
        return $this->_properties['tag_string'];
    }
    if (empty($this->tags)) {
        return '';
    }
    $tags = new Collection($this->tags);
    $str = $tags->reduce(function ($string, $tag) {
        return $string . $tag->title . ', ';
    }, '');
    return trim($str, ', ');
}
```

Isso vai nos deixar acessar a propriedade computada `$bookmark->tag_string`. Vamos usar essa propriedade em inputs mais tarde. Lembre-se de adicionar a propriedade `tag_string` a lista `_accessible` em sua entidade.

Em **src/Model/Entity/Bookmark.php** adicione o `tag_string` ao `_accessible` desta forma:

```
protected $_accessible = [
    'user_id' => true,
    'title' => true,
    'description' => true,
    'url' => true,
    'user' => true,
    'tags' => true,
    'tag_string' => true,
];
```

Atualizando as views

Com a entidade atualizado, podemos adicionar uma nova entrada para as nossas tags. Nas views `add` e `edit`, substitua `tags._ids` pelo seguinte:

```
<?= $this->Form->input('tag_string', ['type' => 'text']) ?>
```

Persistindo a string tag

Agora que podemos ver as tags como uma string existente, vamos querer salvar os dados também. Por marcar o `tag_string` como acessível, o ORM irá copiar os dados do pedido em nossa entidade. Podemos usar um método `beforeSave` para analisar a cadeia tag e encontrar/construir as entidades relacionadas. Adicione o seguinte em **src/Model/Table/BookmarksTable.php**:

```
public function beforeSave($event, $entity, $options)
{
    if ($entity->tag_string) {
        $entity->tags = $this->_buildTags($entity->tag_string);
    }
}
```

```

    }
}

protected function _buildTags($tagString)
{
    $new = array_unique(array_map('trim', explode(',', $tagString)));
    $out = [];
    $query = $this->Tags->find()
        ->where(['Tags.title IN' => $new]);

    // Remove tags existentes da lista de novas tags.
    foreach ($query->extract('title') as $existing) {
        $index = array_search($existing, $new);
        if ($index !== false) {
            unset($new[$index]);
        }
    }
    // Adiciona tags existentes.
    foreach ($query as $tag) {
        $out[] = $tag;
    }
    // Adiciona novas tags.
    foreach ($new as $tag) {
        $out[] = $this->Tags->newEntity(['title' => $tag]);
    }
    return $out;
}

```

Embora esse código seja um pouco mais complicado do que o que temos feito até agora, ele ajuda a mostrar o quão poderosa a ORM do CakePHP é. Você pode facilmente manipular resultados da consulta usando os métodos de *Collections (Coleções)*, e lidar com situações em que você está criando entidades sob demanda com facilidade.

Terminando

Nós expandimos nossa aplicação bookmarker para lidar com situações de autenticação e controle de autorização/acesso básico. Nós também adicionamos algumas melhorias agradáveis à UX, aproveitando os recursos FormHelper e ORM.

Obrigado por dispor do seu tempo para explorar o CakePHP. Em seguida, você pode saber mais sobre o *Models (Modelos)*, ou você pode ler os */topics*.

3.0 - Guia de migração

Esta página resume as alterações do CakePHP 2.x e irá auxiliar na migração do seu projeto para a versão 3.0, e também será uma referência para atualizá-lo quanto às principais mudanças do branch 2.x. Certifique-se de ler também as outras páginas nesse guia para conhecer todas as novas funcionalidades e mudanças na API.

Requerimentos

- O CakePHP 3.x suporta o PHP 5.4.16 e acima.
- O CakePHP 3.x precisa da extensão mbstring.
- O CakePHP 3.x precisa da extensão intl.

Aviso: O CakePHP 3.0 não irá funcionar se você não atender aos requisitos acima.

Ferramenta de atualização

Enquanto este documento cobre todas as alterações e melhorias feitas no CakePHP 3.0, nós também criamos uma aplicação de console para ajudar você a completar mais facilmente algumas das alterações mecânicas que consomem tempo. Você pode [pegar a ferramenta de atualização no GitHub](#)¹⁰.

Layout do diretório da aplicação

O Layout do diretório da aplicação mudou e agora segue o [PSR-4](#)¹¹. Você deve usar o projeto do [esqueleto da aplicação](#)¹² como um ponto de referência quando atualizar sua aplicação.

¹⁰ <https://github.com/cakephp/upgrade>

¹¹ <http://www.php-fig.org/psr/psr-4/>

¹² <https://github.com/cakephp/app>

O CakePHP deve ser instalado via Composer

Como o CakePHP não pode mais ser instalado facilmente via PEAR, ou em um diretório compartilhado, essas opções não são mais suportadas. Ao invés disso, você deve usar o [Composer](http://getcomposer.org)¹³ para instalar o CakePHP em sua aplicação.

Namespaces

Todas as classes do core do CakePHP agora usam namespaces e seguem as especificações de autoloader (auto-carregamento) do PSR-4. Por exemplo `src/Cache/Cache.php` tem o namespace `Cake\Cache\Cache`. Constantes globais e métodos de helpers como `__()` e `debug()` não usam namespaces por questões de conveniência.

Constantes removidas

As seguintes constantes obsoletas foram removidas:

- `IMAGES`
- `CSS`
- `JS`
- `IMAGES_URL`
- `JS_URL`
- `CSS_URL`
- `DEFAULT_LANGUAGE`

Configuração

As configurações no CakePHP 3.0 estão significativamente diferentes que nas versões anteriores. Você deve ler a documentação [Configuração](#) para ver como a configuração é feita.

Você não pode mais usar o `App::build()` para configurar caminhos adicionais de classes. Ao invés disso, você deve mapear caminhos adicionais usando o autoloader da sua aplicação. Veja a seção [Caminhos de Classes Adicionais](#) para mais informações.

Três novas variáveis de configuração fornecem o caminho de configuração para plugins, views e arquivos de localização. Você pode adicionar vários caminhos em `App.paths.templates`, `App.paths.plugins`, `App.paths.locales` para configurar múltiplos caminhos para templates, plugins e arquivos de localização respectivamente.

A chave de configuração `www_root` mudou para `wwwRoot` devido a consistência. Por favor, ajuste seu arquivo de configuração `app.php` assim como qualquer uso de `Configure::read('App.wwwRoot')`.

¹³ <http://getcomposer.org>

Novo ORM

O CakePHP 3.0 possui um novo ORM que foi refeito do zero. O novo ORM é significativamente diferente e incompatível com o anterior. Migrar para o novo ORM necessita de alterações extensas em qualquer aplicação que esteja sendo atualizada. Veja a nova documentação [Models \(Modelos\)](#) para informações de como usar o novo ORM.

Básico

- O `LogError()` foi removido, ele não tinha vantagens e era raramente ou mesmo, nunca usado.
- As seguintes funções globais foram removidas: `config()`, `cache()`, `clearCache()`, `convertSlashes()`, `am()`, `fileExistsInPath()`, `sortByKey()`.

Debug

- A função `Configure::write('debug', $bool)` não suporta mais 0/1/2. Um booleano simples é usado para mudar o modo de debug para ligado ou desligado.

Especificações/Configurações de objetos

- Os objetos usados no CakePHP agora tem um sistema consistente de armazenamento/recuperação de configuração-de-instância. Os códigos que anteriormente acessavam, por exemplo `$object->settings`, devem ser atualizados para usar `$object->config()` alternativamente.

Cache

- Memcache foi removido, use `Cake\Cache\Cache\Engine\Memcached` alternativamente.
- Cache engines são carregados sob demanda no primeiro uso.
- `Cake\Cache\Cache::engine()` foi adicionado.
- `Cake\Cache\Cache::enabled()` foi adicionado. Substituindo a opção de configuração `Cache.disable`.
- `Cake\Cache\Cache::enable()` foi adicionado.
- `Cake\Cache\Cache::disable()` foi adicionado.
- Configuração de cache agora é imutável. Se você precisa alterar a configuração, será necessário desfazer-se da configuração e recriá-la. Isso previne problemas de sincronização com as opções de configuração.
- `Cache::set()` foi removido. É recomendado criar múltiplas configurações de cache para substituir ajustes de configuração em tempo de execução anteriormente possíveis com `Cache::set()`.
- Todas as subclasses `CacheEngine` agora implementam um método `config()`.
- `Cake\Cache\Cache::readMany()`, `Cake\Cache\Cache::deleteMany()`, e `Cake\Cache\Cache::writeMany()` foram adicionados.

Todos os métodos `Cake\Cache\Cache\CacheEngine` agora são responsáveis por manipular o prefixo chave configurado. O `Cake\Cache\CacheEngine::write()` não mais permite definir a duração na escrita, a duração é captada pela configuração de tempo de execução do mecanismo de cache. Chamar um método cache com uma chuva vazia irá lançar uma `InvalidArgumentException` ao invés de retornar `false`.

Core

App

- `App::pluginPath()` foi removido. Use `CakePlugin::path()` alternativamente.
- `App::build()` foi removido.
- `App::location()` foi removido.
- `App::paths()` foi removido.
- `App::load()` foi removido.
- `App::objects()` foi removido.
- `App::RESET` foi removido.
- `App::APPEND` foi removido.
- `App::PREPEND` foi removido.
- `App::REGISTER` foi removido.

Plugin

- O `Cake\Core\Plugin::load()` não configura a carga automática a menos que você defina a opção `autoload` como `true`.
- Quanto estiver carregando plugins você não pode mais fornecer um callable.
- Quanto estiver carregando plugins você não pode mais fornecer um array de arquivos de configuração para carregar.

Configure

- O `Cake\Configure\PhpReader` foi renomeado para `Cake\Core\Configure\EnginePhpConfig`
- O `Cake\Configure\IniReader` foi renomeado para `Cake\Core\Configure\EngineIniConfig`
- O `Cake\Configure\ConfigReaderInterface` foi renomeado para `Cake\Core\Configure\ConfigEngineInterface`
- O `Cake\Core\Configure::consume()` foi adicionado.
- O `Cake\Core\Configure::load()` agora espera o nome de arquivo sem o sufixo de extensão como isso pode ser derivado do mecanismo. Ex.: para usar o `PhpConfig` use `app` para carregar `app.php`.
- Definir uma variável `$config` no arquivo PHP `config` está obsoleto. `Cake\Core\Configure\EnginePhpConfig` agora espera que o arquivo de configuração retorne um array.
- Um novo mecanismo de configuração `Cake\Core\Configure\EngineJsonConfig` foi adicionado.

Object

A classe `Object` foi removida. Ela anteriormente continha um monte de métodos que eram utilizados em vários locais no framework. O mais útil destes métodos foi extraído como um `trait`. Você pode usar o `Cake\Log\LogTrait` para acessar o método `log()`. O `Cake\Routing\RequestActionTrait` fornece o método `requestAction()`.

Console

O executável `cake` foi movido do diretório `app/Console` para o diretório `bin` dentro do esqueleto da aplicação. Você pode agora invocar o console do CakePHP com `bin/cake`.

TaskCollection Substituído

Essa classe foi renomeada para `Cake\Console\TaskRegistry`. Veja a seção em *Objetos de Registro* para mais informações sobre funcionalidades fornecidas pela nova classe. Você pode usar o `cake upgrade rename_collections` para ajuda ao atualizar seu código. Tarefas não tem mais acesso a callbacks, como nunca houve nenhum callback para se usar.

Shell

- O `Shell::__construct()` foi alterado. Ele agora usa uma instância de `Cake\Console\ConsoleIo`.
- O `Shell::param()` foi adicionado como um acesso conveniente aos parâmetros.

Adicionalmente todos os métodos shell serão transformados em camel case quando invocados. Por exemplo, se você tem um método `hello_world()` dentro de um shell e chama ele com `bin/cake my_shell hello_world`, você terá que renomear o método para `helloWorld`. Não há necessidade de mudanças no modo que você chama os métodos/comandos.

ConsoleOptionParser

- O `ConsoleOptionParser::merge()` foi adicionado para mesclar os parsers.

ConsoleInputArgument

- O `ConsoleInputArgument::isEqualTo()` foi adicionado para comparar dois argumentos.

Shell / Tarefa

Os Shells e Tarefas foram movidas de `Console/Command` e `Console/Command/Task` para `Shell` e `Shell/Task`, respectivamente.

ApiShell Removido

O ApiShell foi removido pois ele não fornecia nenhum benefício além do próprio arquivo fonte e da documentação/API¹⁴ online.

SchemaShell Removido

O SchemaShell foi removido como ele nunca foi uma implementação completa de migração de banco de dados e surgiram ferramentas melhores como o Phinx¹⁵. Ele foi substituído pelo CakePHP Migrations Plugin¹⁶ que funciona como um empacotamento entre o CakePHP e o Phinx¹⁷.

ExtractTask

- O `bin/cake i18n extract` não inclui mais mensagens de validação sem tradução. Se você quiser mensagens de validação traduzidas você deve encapsula-las com chamadas `__()` como qualquer outro conteúdo.

BakeShell / TemplateTask

- O Bake não faz mais parte do fonte do núcleo e é suplantado pelo CakePHP Bake Plugin¹⁸
- Os templates do Bake foram movidos para `src/Template/Bake`.
- A sintaxe dos templates do Bake agora usam tags estilo erb (`<% %>`) para denotar lógica de template, permitindo código php ser tratado como texto plano.
- O comando `bake view` foi renomeado para `bake template`.

Eventos

O método `getEventManager()`, foi removido de todos os objetos que continham. Um método `eventManager()` é agora fornecido pelo `EventManagerTrait`. O `EventManagerTrait` contém a lógica de instanciação e manutenção de uma referência para um gerenciador local de eventos.

O subsistema `Event` teve um monte de funcionalidades opcionais removidas. Quando despachar eventos você não poderá mais usar as seguintes opções:

- `passParams` Essa opção está agora ativada sempre implicitamente. Você não pode desliga-la.
- `break` Essa opção foi removida. Você deve agora parar os eventos.
- `breakOn` Essa opção foi removida. Você deve agora parar os eventos.

Log

- As configurações do Log agora não imutáveis. Se você precisa alterar a configuração você deve primeiro derrubar a configuração e então recria-la. Isso previne problemas de sincronização com opções de configuração.

¹⁴ <https://api.cakephp.org/>

¹⁵ <https://phinx.org/>

¹⁶ <https://github.com/cakephp/migrations>

¹⁷ <https://phinx.org/>

¹⁸ <https://github.com/cakephp/bake>

- Os mecanismos de Log agora são carregados tardiamente após a primeira escrita nos logs.
- O `Cake\Log\Log::engine()` foi adicionado.
- Os seguintes métodos foram removidos de `Cake\Log\Log::defaultLevels(), enabled(), enable(), disable()`.
- Você não pode mais criar níveis personalizados usando `Log::levels()`.
- Quando configurar os loggers você deve usar 'levels' ao invés de 'types'.
- Você não pode mais especificar níveis personalizados de log. Você deve usar o conjunto padrão de níveis de log. Você deve usar escopos de log para criar arquivos de log personalizados ou manipulações específicas para diferentes seções de sua aplicação. Usando um nível de log não padrão irá lançar uma exceção.
- O `Cake\Log\LogTrait` foi adicionado. Você pode usar este trait em suas classes para adicionar o método `log()`.
- O escopo de log passado para `Cake\Log\Log::write()` é agora encaminhado para o método `write()` dos mecanismos de log de maneira a fornecer um melhor contexto para os mecanismos.
- Os mecanismos de Log agora são necessários para implementar `Psr\Log\LogInterface` invés do próprio `LogInterface` do Cake. Em geral, se você herdou o `Cake\Log\Engine\BaseEngine` você só precisa renomear o método `write()` para `log()`.
- O `Cake\Log\Engine\FileLog` agora grava arquivos em `ROOT/logs` no lugar de `ROOT/tmp/logs`.

Roteamento

Parâmetros Nomeados

Os parâmetros nomeados foram removidos no 3.0. Os parâmetros nomeados foram adicionados no 1.2.0 como uma versão 'bonita' de parâmetros de requisição. Enquanto o benefício visual é discutível, os problemas criados pelos parâmetros nomeados não são.

Os parâmetros nomeados necessitam manipulação especial no CakePHP assim como em qualquer biblioteca PHP ou JavaScript que necessite interagir com eles, os parâmetros nomeados não são implementados ou entendidos por qualquer biblioteca *exceto* o CakePHP. A complexidade adicionada e o código necessário para dar suporte aos parâmetros nomeados não justificam a sua existência, e eles foram removidos. No lugar deles, você deve agora usar o padrão de parâmetros de requisição (querystring) ou argumentos passados configurados nas rotas. Por padrão o Router irá tratar qualquer parâmetro adicional ao `Router::url()` como argumentos de requisição.

Como muitas aplicações ainda precisarão analisar URLs contendo parâmetros nomeados, o `Cake\Routing\Router::parseNamedParams()` foi adicionado para permitir compatibilidade com URLs existentes.

RequestActionTrait

- O `Cake\Routing\RequestActionTrait::requestAction()` teve algumas de suas opções extras alteradas:
 - o `options[url]` é agora `options[query]`.
 - o `options[data]` é agora `options[post]`.
 - os parâmetros nomeados não são mais suportados.

Roteador

- Os parâmetros nomeados foram removidos, veja acima para mais informações.
- A opção `full_base` foi substituída com a opção `_full`.
- A opção `ext` foi substituída com a opção `_ext`.
- As opções `_scheme`, `_port`, `_host`, `_base`, `_full`, `_ext` foram adicionadas.
- As URLs em strings não são mais modificados pela adição de plugin/controller/nomes de prefixo.
- A manipulação da rota padrão de `fallback` foi removida. Se nenhuma rota combinar com o conjunto de parâmetros, o `/` será retornado.
- As classes de rota são responsáveis por *toda* geração de URLs incluindo parâmetros de requisição (query string). Isso faz com que as rotas sejam muito mais poderosas e flexíveis.
- Parâmetros persistentes foram removidos. Eles foram substituídos pelo `Cake\Routing\Router::urlFilter()` que permite um jeito mais flexível para mudar URLs sendo roteadas reversamente.
- O `Router::parseExtensions()` foi removido. Use o `Cake\Routing\Router::extensions()` no lugar. Esse método **deve** ser chamado antes das rotas serem conectadas. Ele não irá modificar rotas existentes.
- O `Router::setExtensions()` foi removido. Use o `Cake\Routing\Router::extensions()` no lugar.
- O `Router::resourceMap()` foi removido.
- A opção `[method]` foi renomeada para `_method`.
- A habilidade de combinar cabeçalhos arbitrários com parâmetros no estilo `[]` foi removida. Se você precisar combinar/analisar em condições arbitrárias considere usar classes personalizadas de roteamento.
- O `Router::promote()` foi removido.
- O `Router::parse()` irá agora lançar uma exceção quando uma URL não puder ser atendida por nenhuma rota.
- O `Router::url()` agora irá lançar uma exceção quando nenhuma rota combinar com um conjunto de parâmetros.
- Os escopos de rotas foram adicionados. Escopos de rotas permitem você manter seu arquivo de rotas limpo e dar dicas de rotas em como otimizar análise e reversão de rotas de URL.

Route

- O `CakeRoute` foi renomeado para `Route`.
- A assinatura de `match()` mudou para `match($url, $context = [])`. Veja `Cake\Routing\Route::match()` para mais informações sobre a nova assinatura.

Configuração de Filtros do Despachante Mudaram

Os filtros do despachante não são mais adicionados em sua aplicação usando o `Configure`. Você deve agora anexa-los com `Cake\Routing\DispatcherFactory`. Isso significa que sua aplicação usava `Dispatcher.filters`, você deve usar agora o método `Cake\Routing\DispatcherFactory::add()`.

Além das mudanças de configuração, os filtros do despachante tiveram algumas convenções atualizadas e novas funcionalidades. Veja a documentação em [Filtros do Dispatcher](#) para mais informações.

FilterAssetFilter

- Os itens de plugins e temas manipulados pelo AssetFilter não são mais lidos via `include`, ao invés disso eles são tratados como arquivos de texto plano. Isso corrige um número de problemas com bibliotecas javascript como TinyMCE e ambientes com `short_tags` ativadas.
- O suporte para a configuração `Asset.filter` e ganchos foram removidos. Essa funcionalidade pode ser facilmente substituída com um plugin ou filtro de despachante.

Rede

Requisição

- O `CakeRequest` foi renomeada para `Cake\Network\Request`.
- O `Cake\Network\Request::port()` foi adicionado.
- O `Cake\Network\Request::scheme()` foi adicionado.
- O `Cake\Network\Request::cookie()` foi adicionado.
- O `Cake\Network\Request::$trustProxy` foi adicionado. Isso torna mais fácil colocar aplicações CakePHP atrás de balanceadores de carga.
- O `Cake\Network\Request::$data` não é mais mesclado com a chave de dados prefixada, pois esse prefixo foi removido.
- O `Cake\Network\Request::env()` foi adicionado.
- O `Cake\Network\Request::acceptLanguage()` mudou de um método estático para não-estático.
- O detector de requisição para dispositivos móveis foi removido do núcleo. Agora o app template adiciona detectores para dispositivos móveis usando a biblioteca `MobileDetect`.
- O método `onlyAllow()` foi renomeado para `allowMethod()` e não aceita mais “argumentos var”. Todos os nomes de métodos precisam ser passados como primeiro argumento, seja como string ou como array de strings.

Resposta

- O mapeamento do `mimetype text/plain` para extensão `csv` foi removido. Como consequência o `Cake\Controller\Component\RequestHandlerComponent` não define a extensão para `csv` se o cabeçalho `Accept` tiver o `mimetype text/plain` que era um problema comum quando recebia uma requisição XHR do jQuery.

Sessões

A classe de sessão não é mais estática, agora a sessão (`session`) pode ser acessada através do objeto de requisição (`request`). Veja a documentação em [Sessions](#) para ver como usar o objeto de sessão.

- O `Cake\Network\Session` e classes de sessão relacionadas foram movidas para o namespace `Cake\Network`.
- O `SessionHandlerInterface` foi removido em favor ao fornecido pelo próprio PHP.
- A propriedade `Session::$requestCountdown` foi removida.

- O funcionalidade de sessão `checkAgent` foi removida. Ela causava um monte de bugs quando quadros do chrome e o flash player estavam envolvidos.
- A convenção de nome para a tabela de sessão no banco de dados agora é `sessions` ao invés de `cake_sessions`.
- O cookie de tempo limite da sessão é atualizado automaticamente em conjunto com o tempo limite dos dados de sessão.
- O caminho padrão para o cookie de sessão agora é o caminho base da aplicação, ao invés de `/`. Além disso, uma nova variável de configuração `Session.cookiePath` foi adicionada para facilitar a personalização do caminho para os cookies.
- Um novo método conveniente `Cake\Network\Session::consume()` foi adicionado para permitir a leitura e exclusão de dados de sessão em um único passo.
- O valor padrão do argumento `$renew` de `Cake\Network\Session::clear()` mudou de `true` para `false`.

Network\Http

- O `HttpSocket` agora é `Cake\Network\Http\Client`.
- O `HttpClient` foi reescrito do zero. Ele tem uma API mais simples/fácil de usar, suporta novos sistemas de autenticação como OAuth, e uploads de arquivos. Ele usa as API de stream do PHP de modo que não há requirement para o cURL. Veja a documentação [Http Client](#) para mais informações.

Network>Email

- O `Cake\Network>Email>Email::config()` agora é usado para definir perfis de configuração. Isso substitui as classes `EmailConfig` nas versões anteriores.
- O `Cake\Network>Email>Email::profile()` substitui o `config()` como modo de modificar opções de configuração por instância.
- O `Cake\Network>Email>Email::drop()` foi adicionado para permitir a remoção de configurações de email.
- O `Cake\Network>Email>Email::configTransport()` foi adicionado para permitir a definição de configurações de transporte. Essa mudança retira as opções de transporte dos perfis de entrega e permite a você reusar facilmente os transportes através de perfis de e-mails.
- O `Cake\Network>Email>Email::dropTransport()` foi adicionado para permitir a remoção de configurações de transporte.

Controller

Controller

- As propriedades `$helpers` e `$components` agora estão mescladas com **todas** classes pai, não apenas a `AppController` e o plugin de `AppController`. As propriedades são mescladas de modo diferente agora também. No lugar de todas as configurações em todas as classes serem mescladas juntas, as configurações definidas nas classes filho serão usadas. Isso quer dizer que se você tem alguma configurações definida no seu

AppController, e alguma configuração definida em uma a subclasse, apenas a configuração na subclasse será usada.

- O `Controller::httpCodes()` foi removido, use o `Cake\Network\Response::httpCodes()` no lugar.
- O `Controller::disableCache()` foi removido, use o `Cake\Network\Response::disableCache()` no lugar.
- O `Controller::flash()` foi removido. Esse método era raramente usado em aplicações reais e não tinha mais propósito algum.
- O `Controller::validate()` e `Controller::validationErrors()` foram removidos. Eles eram restos dos dias do 1.x onde as preocupações com os models + controllers eram muito mais entrelaçados.
- O `Controller::loadModel()` agora carrega uma tabela de objetos.
- A propriedade `Controller::$scaffold` foi removida. O scaffolding dinâmico foi removido do núcleo do CakePHP. Um plugin de scaffolding melhorado, chamado CRUD, pode ser encontrado em: <https://github.com/FriendsOfCake/crud>
- A propriedade `Controller::$ext` foi removida. Você deve agora estender e sobrescrever a propriedade `View::$_ext` se você deseja usar uma extensão de arquivo de visão não padrão.
- A propriedade `Controller::$methods` foi removida. Você deve usar o `Controller::isAction()` para determinar quando ou não um nome de método é uma ação. Essa mudança foi feita para permitir personalizações mais fáceis do que vai contar ou não como uma ação.
- A propriedade `Controller::$Components` foi removida e substituída pelo `_components`. Se você precisar carregar componentes em tempo de execução você deve usar o `$this->loadComponent()` em seu controller.
- A assinatura do `Cake\Controller\Controller::redirect()` mudou para `Controller::redirect(string|array $url, int $status = null)`. O terceiro argumento `$exit` foi removido. O método não pode mais enviar resposta e sair do script, no lugar ele retorna uma instância de `Response` com os cabeçalhos apropriados definidos.
- As propriedades mágicas `base`, `webroot`, `here`, `data`, `action`, e `params` foram removidas. Você deve acessar todas essas propriedades em `$this->request` no lugar.
- Métodos de controlar prefixados com sublinhado como `_someMethod()` não são mais tratados como métodos privados. Use as palavras chaves de visibilidade apropriadas no lugar. Somente métodos públicos podem ser usados como ação de controllers.

Scaffold Removido

O scaffolding dinâmico no CakePHP foi removido do núcleo do CakePHP. Ele não era usado com frequência, e não era voltado para uso em produção. Um plugin melhorado de scaffolding, chamado CRUD, pode ser encontrado em: <https://github.com/FriendsOfCake/crud>

ComponentCollection Substituído

Essa classe foi renomeada para `Cake\Controller\ComponentRegistry`. Veja a seção em *Objetos de Registro* para mais informações sobre as funcionalidades fornecidas pela nova classe. Você pode usar o `cake upgrade rename_collections` para ajudar você a atualizar o seu código.

Components

- A propriedade `_Collection` é agora `_registry`. Ela contém uma instância do `Cake\Controller\ComponentRegistry` agora.
- Todos components devem agora usar o método `config()` para obter/definir configurações.
- A configuração padrão para components deve ser definido na propriedade `$_defaultConfig`. Essa propriedade é automaticamente mesclada com qualquer configuração fornecida pelo construtor.
- Opções de configuração não são mais definidas como propriedades públicas.
- O método `Component::initialize()` não é mais um event listener (ouvinte de eventos). Ao invés disso, ele é um gancho pós-construtor como o `Table::initialize()` e `Controller::initialize()`. O novo método `Component::beforeFilter()` é ligado ao mesmo evento que o `Component::initialize()` costumava ser. O método de inicialização deve ter a seguinte assinatura `initialize(array $config)`.

Controller\Components

CookieComponent

- Ele usa o `Cake\Network\Request::cookie()` para ler os dados de cookies, isso facilita os testes, e permite o `ControllerTestCase` definir os cookies.
- Os Cookies encriptados pelas versões anteriores do CakePHP usando o método `cipher()`, agora não podem ser lidos, pois o `Security::cipher()` foi removido. Você precisará reencriptar os cookies com o método `rijndael()` ou `aes()` antes de atualizar.
- O `CookieComponent::type()` foi removido e substituído com dados de configuração acessados através de `config()`.
- O `write()` não aceita mais os parâmetros `encryption` ou `expires`. Os dois agora são gerenciados através de dados de configuração. Veja [CookieComponent](#) para mais informações.
- O caminho padrão para os cookies agora é o caminho base da aplicação, ao invés de `/`.

AuthComponent

- O `Default` é agora o hasher de senhas padrão usado pelas classes de autenticação. Ele usa exclusivamente o algoritmo de hash `bcrypt`. Se você deseja continuar usando o hash `SHA1` usado no 2.x, use `'passwordHasher' => 'Weak'` nas configurações de seu autenticador.
- O novo `FallbackPasswordHasher` foi adicionado para ajudar os usuários migrar senhas antigas de um algoritmo para o outro. Veja a documentação do `AuthComponent` para mais informações.
- A classe `BlowfishAuthenticate` foi removida. Apenas use `FormAuthenticate`.
- A classe `BlowfishPasswordHasher` foi removida. Use o `DefaultPasswordHasher` no lugar.
- O método `loggedIn()` foi removido. Use o `user()` no lugar.
- As opções de configuração não são mais definidas como propriedades públicas.
- Os métodos `allow()` e `deny()` não aceitam mais `"var args"`. Todos os nomes de métodos precisam ser passados como primeiro argumento, seja como string ou array de strings.

- O método `login()` foi removido e substituído por `setUser()`. Para logar um usuário agora você deve chamar `identify()` que retorna as informações do usuário caso identificado com sucesso e então usar `setUser()` para salvar as informações na sessão de maneira persistente entre as requisições.
- O `BaseAuthenticate::_password()` foi removido. Use a classe `PasswordHasher` no lugar.
- O `BaseAuthenticate::logout()` foi removido.
- O `AuthComponent` agora dispara dois eventos `Auth.afterIdentify` e `Auth.logout` após um usuário ser identificado e antes de um usuário ser deslogado respectivamente. Você pode definir funções de callback para esses eventos retornando um array mapeado no método `implementedEvents()` de sua classe de autenticação.

Classes relacionadas a ACL foram movidas para um plugin separado. Hashers de senha, fornecedores de Autenticação e Autorização foram movidos para o namespace `\Cake\Auth`. Você DEVE mover seus fornecedores e hashers para o namespace `App\Auth` também.

RequestHandlerComponent

- Os seguintes métodos foram removidos do componente `RequestHandler`: `isAjax()`, `isFlash()`, `isSSL()`, `isPut()`, `isPost()`, `isGet()`, `isDelete()`. Use o método `Cake\Network\Request::is()` no lugar com o argumento relevante.
- O `RequestHandler::setContent()` foi removido, use `Cake\Network\Response::type()` no lugar.
- O `RequestHandler::getReferer()` foi removido, use `Cake\Network\Request::referer()` no lugar.
- O `RequestHandler::getClientIP()` foi removido, use `Cake\Network\Request::clientIp()` no lugar.
- O `RequestHandler::getAjaxVersion()` foi removido.
- O `RequestHandler::mapType()` foi removido, use `Cake\Network\Response::mapType()` no lugar.
- As opções de configuração não são mais definidas como propriedades públicas.

SecurityComponent

- Os seguintes métodos e as propriedades relacionadas foram removidas do componente `Security`: `requirePost()`, `requireGet()`, `requirePut()`, `requireDelete()`. Use o `Cake\Network\Request::allowMethod()` no lugar.
- `SecurityComponent::$disabledFields()` foi removido, use o `SecurityComponent::$unlockedFields()`.
- As funções relacionadas ao CSRF no `SecurityComponent` foram extraídas e movidas em separado no `CsrfComponent`. Isso permite que você use a proteção CSRF facilmente sem ter que usar prevenção de adulteração de formulários.
- As opções de configuração não são mais definidas como propriedades públicas.
- Os métodos `requireAuth()` e `requireSecure()` não aceitam mais “var args”. Todos os nomes de métodos precisam ser passados como primeiro argumento, seja como string ou array de strings.

SessionComponent

- O `SessionComponent::setFlash()` está obsoleto. Você deve usar o *Flash* no lugar.

Error

ExceptionRenderers personalizados agora espera-se que retornem ou um objeto `Cake\Network\Response` ou uma string quando renderizando erros. Isso significa que qualquer método que manipule exceções específicas devem retornar uma resposta ou valor de string.

Model

A camada de model do 2.x foi completamente reescrita e substituída. Você deve revisar o *Guia de atualização para o novo ORM* para saber como usar o novo ORM.

- A classe `Model` foi removida.
- A classe `BehaviorCollection` foi removida.
- A classe `DboSource` foi removida.
- A classe `Datasource` foi removida.
- As várias classes de fonte de dados foram removidas.

ConnectionManager

- O `ConnectionManager` (gerenciador de conexão) foi movido para o namespace `Cake\Datasource`.
- O `ConnectionManager` teve os seguintes métodos removidos:
 - `sourceList`
 - `getSourceName`
 - `loadDataSource`
 - `enumConnectionObjects`
- O `Database\ConnectionManager::config()` foi adicionado e é agora o único jeito de configurar conexões.
- O `Database\ConnectionManager::get()` foi adicionado. Ele substitui o `getDataSource()`.
- O `Database\ConnectionManager::configured()` foi adicionado. Ele junto com `config()` substitui o `sourceList()` e `enumConnectionObjects()` com uma API mais padrão e consistente.
- O `ConnectionManager::create()` foi removido. Ele pode ser substituído por `config($name, $config)` e `get($name)`.

Behaviors

- Os métodos de comportamentos (behaviors) prefixados com sublinhado como `_someMethod()` não são mais tratados como métodos privados. Use as palavras chaves de visibilidade.

TreeBehavior

O TreeBehavior foi completamente reescrito para usar o novo ORM. Embora ele funcione do mesmo modo que no 2.x, alguns métodos foram renomeados ou removidos:

- `TreeBehavior::children()` é agora uma busca personalizada `find('children')`.
- `TreeBehavior::generateTreeList()` é agora uma busca personalizada `find('treeList')`.
- `TreeBehavior::getParentNode()` foi removido.
- `TreeBehavior::getPath()` é agora uma busca personalizada `find('path')`.
- `TreeBehavior::reorder()` foi removido.
- `TreeBehavior::verify()` foi removido.

Suíte de Testes

Casos de Teste

- O `_normalizePath()` foi adicionado para permitir testes de comparação de caminhos para executar em todos os sistemas operacionais, independente de sua configuração (\ no Windows vs / no UNIX, por exemplo).

Os seguintes métodos de asserção foram removidos já que eles estavam há muito obsoletos e foram substituídos pelo seu equivalente no PHPUnit:

- `assertEqual()` é substituído por `assertEquals()`
- `assertNotEqual()` é substituído por `assertNotEquals()`
- `assertIdentical()` é substituído por `assertSame()`
- `assertNotIdentical()` é substituído por `assertNotSame()`
- `assertPattern()` é substituído por `assertRegExp()`
- `assertNoPattern()` é substituído por `assertNotRegExp()`
- `assertReference()` é substituído por `assertSame()`
- `assertIsA()` é substituído por `assertInstanceOf()`

Note que alguns métodos tiveram a ordem dos argumentos trocada, ex. `assertEqual($is, $expected)` deve ser agora `assertEquals($expected, $is)`.

Os seguintes métodos de asserção estão obsoletos e serão removidos no futuro:

- `assertWithinMargin()` é substituído por `assertWithinRange()`
- `assertTags()` é substituído por `assertHtml()`

Em ambas as substituições dos métodos também mudaram a ordem dos argumentos para manter a consistência na API com `$expected` como primeiro argumento.

Os seguintes métodos de asserção foram adicionados:

- `assertNotWithinRange()` em contrapartida ao `assertWithinRange()`

View

Temas são agora Plugins Básicos

Ter os temas e plugins de modo a criar components modulares da aplicação se provou limitado e confuso. No CakePHP 3.0, temas não residem mais **dentro** da aplicação. Ao invés disso, eles são plugins independentes. Isso resolveu alguns problemas com temas:

- Você não podia colocar temas *nos* plugins.
- Temas não podiam fornecer helpers (helpers), ou classes de visão personalizadas.

Esses dois problemas foram resolvidos ao converter os temas em plugins.

Pasta das views renomeada

As pastas contendo os arquivos de views agora ficam em **src/Template** no lugar de **src/View**. Isso foi feito para separar os arquivos de visão dos arquivos contendo classes php. (ex. helpers, Classes de visão).

As seguintes pastas de Visão foram renomeadas para evitar colisão de nomes com nomes de controllers:

- Layouts agora é Layout
- Elements agora é Element
- Errors agora é Error
- Emails agora é Email (o mesmo para Email dentro de Layout)

Coleção de Helpers Substituída

Essa classe foi renomeada para `Cake\View\HelperRegistry`. Veja a seção em *Objetos de Registro* para mais informações sobre as funcionalidades fornecidas pela nova classe. Você pode usar o `cake upgrade rename_collections` para ajudar você a atualizar seu código.

Classe View

- A chave `plugin` foi removida do argumento `$options` de `Cake\View\View::element()`. Especifique o nome do elemento como `AlgunPlugin.nome_do_elemento` no lugar.
- O `View::getVar()` foi removido, use o `Cake\View\View::get()` no lugar.
- O `View::$ext` foi removido e no lugar uma propriedade protegida `View::$_ext` foi adicionada.
- O `View::addScript()` foi removido. Use o *Usando View Blocks* no lugar.
- As propriedades mágicas `base`, `webroot`, `here`, `data`, `action`, e `params` foram removidas. Ao invés disso, você deve acessar todas essas propriedades no `$this->request`.
- O `View::start()` não se liga mais a um bloco existente. Ao invés disso ele irá sobrescrever o conteúdo do bloco quando o `end()` for chamado. Se você precisa combinar o conteúdo de um bloco você deverá buscar o conteúdo do bloco quando chamar o `start` uma segunda vez, ou usar o modo de captura de `append()`.
- O `View::prepend()` não tem mais um modo de captura.
- O `View::startIfEmpty()` foi removido. Agora que o `start()` sempre sobrescreve, o `startIfEmpty` não tem mais propósito.

- A propriedade `View::$Helpers` foi removida e substituída com `_helpers`. Se você precisar carregar helpers em tempo de execução você deve usar o `$this->addHelper()` em seus arquivos de visão.
- O View agora irá lançar `Cake\View\Exception\MissingTemplateException` quando templates estiverem faltando, ao invés de `MissingViewException`.

ViewBlock

- O `ViewBlock::append()` foi removido, use o `Cake\View\ViewBlock::concat()` no lugar. Entretanto o `View::append()` ainda existe.

JsonView

- Agora os dados JSON terão as entidades HTML codificadas por padrão. Isso previne possíveis problemas de XSS quando o conteúdo de visão JSON está encapsulado em arquivos HTML.
- O `Cake\View\JsonView` agora suporta a variável de visão `_jsonOptions`. Isso permite a você configurar as opções de máscara de bits usadas ao gerar JSON.

XmlView

- A `Cake\View\XmlView` agora suporta a variável de visão `_xmlOptions`. Isso permite a você configurar as opções usadas quando gerar XML.

View\Helper

- A propriedade `$settings` é agora chamada `$_config` e deve ser acessada através do método `config()`.
- As opções de configuração não são mais definidas como propriedades públicas.
- O `Helper::clean()` foi removido. Ele nunca foi robusto o suficiente para prevenir completamente XSS. Ao invés disso você deve escapar o conteúdo com `h` ou ou usar uma biblioteca dedicada como o `htmlPurifier`.
- O `Helper::output()` foi removido. Esse método estava obsoleto no 2.x.
- Os métodos `Helper::webroot()`, `Helper::url()`, `Helper::assetUrl()`, `Helper::assetTimestamp()` foram movidos para o novo ajudante `Cake\View\Helper\UrlHelper`. O `Helper::url()` está agora disponível como `Cake\View\Helper\UrlHelper::build()`.
- Os Assessores Mágicos a propriedades obsoletas foram removidos. A seguinte propriedade agora deve ser acessada a partir do objeto de requisição:
 - base
 - here
 - webroot
 - data
 - action
 - params

Helpers

A classe `Helper` teve os seguintes métodos removidos:

- `Helper::setEntity()`
- `Helper::entity()`
- `Helper::model()`
- `Helper::field()`
- `Helper::value()`
- `Helper::_name()`
- `Helper::_initInputField()`
- `Helper::_selectedArray()`

Esses métodos eram partes usadas apenas pelo `FormHelper`, e parte de uma funcionalidade de persistência de campos que se mostrou problemática com o tempo. O `FormHelper` não precisa mais destes métodos e a complexidades que eles provêm não é mais necessária.

Os seguintes métodos foram removidos:

- `Helper::_parseAttributes()`
- `Helper::_formatAttribute()`

Esses métodos podem agora ser encontrados na classe `StringTemplate` que os helpers usam com frequência. Veja o `StringTemplateTrait` para um jeito fácil de integrar os templates de string em seus próprios helpers.

FormHelper

O `FormHelper` foi completamente reescrito para o 3.0. Ele teve algumas grandes mudanças:

- O `FormHelper` trabalha junto com o novo ORM. Mas também possui um sistema extensível para integrar com outros ORMs e fontes de dados.
- O `FormHelper` possui um sistema de widgets extensível que permite a você criar novos widgets de entrada personalizados e expandir facilmente aqueles inclusos no framework.
- Os Templates de String são a fundação deste ajudante. Ao invés de encher de arrays por toda parte, a maioria do HTML que o `FormHelper` gera pode ser personalizado em um lugar central usando conjuntos de templates.

Além dessas grandes mudanças, foram feitas algumas mudanças menores que causaram rompendo algumas coisas da versão anterior. Essas mudanças devem simplificar o HTML que o `FormHelper` gera e reduzir os problemas que as pessoas tinham no passado:

- O prefixo `data[` foi removido de todas as entradas geradas. O prefixo não tem mais propósito.
- Os vários métodos de entradas independentes, como `text()`, `select()` e outros, não geram mais atributos `id`.
- A opção `inputDefaults` foi removida de `create()`.
- As opções `default` e `onsubmit` do `create()` foram removidas. No lugar você deve usar JavaScript event binding ou definir todos os códigos js necessários para o `onsubmit`.
- O `end()` não gerará mais botões. Você deve criar botões com `button()` ou `submit()`.
- O `FormHelper::tagIsInvalid()` foi removido. Use `isFieldError()` no lugar.

- O `FormHelper::inputDefaults()` foi removido. Você pode usar `templates()` para definir/expandir os templates que o `FormHelper` usa.
- As opções `wrap` e `class` foram removidas do método `error()`.
- A opção `showParents` foi removida do `select()`.
- As opções `div`, `before`, `after`, `between` e `errorMessage` foram removidas do `input()`. Você pode usar templates para atualizar o HTML envoltório. A opção `templates` permite você sobrescrever os templates carregados para uma entrada.
- As opções `separator`, `between`, e `legend` foram removidas do `radio()`. Você pode usar templates para mudar o HTML envoltório agora.
- O parâmetro `format24Hours` foi removido de `hour()`. Ele foi substituído pela opção `format`.
- Os parâmetros `minYear` e `maxYear` foram removidos do `year()`. Ambos podem ser fornecidos como opções.
- Os parâmetros `dateFormat` e `timeFormat` foram removidos do `datetime()`. Você pode usar o template para definir a ordem que as entradas devem ser exibidas.
- O `submit()` teve as opções `div`, `before` e `after` removidas. Você pode personalizar o template `submitContainer` para modificar esse conteúdo.
- O método `inputs()` não aceita mais `legend` e `fieldset` no parâmetro `$fields`, você deve usar o parâmetro `$options`. Ele também exige que o parâmetro `$fields` seja um array. O parâmetro `$blacklist` foi removido, a funcionalidade foi substituída pela especificação de `'field' => false` no parâmetro `$fields`.
- O parâmetro `inline` foi removido do método `postLink()`. Você deve usar a opção `block` no lugar. Definindo `block => true` irá emular o comportamento anterior.
- O parâmetro `timeFormat` para `hour()`, `time()` e `dateTime()` agora é 24 por padrão, em cumprimento ao ISO 8601.
- O argumento `$confirmMessage` de `Cake\View\Helper\FormHelper::postLink()` foi removido. Você deve usar agora a chave `confirm` no `$options` para especificar a mensagem.
- As entradas do tipo `Checkbox` e `radio` são agora renderizadas *dentro* de elementos do tipo `label` por padrão. Isso ajuda a aumentar a compatibilidade com bibliotecas CSS populares como [Bootstrap](http://getbootstrap.com/)¹⁹ e [Foundation](http://foundation.zurb.com/)²⁰.
- As tags de template agora são todas camelBacked (primeira letra minúscula e início de novas palavras em maiúsculo). As tags pré-3.0 `formstart`, `formend`, `hiddenblock` e `inputsubmit` são agora `formStart`, `formEnd`, `hiddenBlock` e `inputSubmit`. Certifique-se de alterá-las se elas estiverem personalizando sua aplicação.

É recomendado que você revise a documentação [Form](#) para mais detalhes sobre como usar o `FormHelper` no 3.0.

HtmlHelper

- O `HtmlHelper::useTag()` foi removido, use `tag()` no lugar.
- O `HtmlHelper::loadConfig()` foi removido. As tags podem ser personalizadas usando `templates()` ou as configurações de templates.
- O segundo parâmetro `$options` para `HtmlHelper::css()` agora sempre irá exigir um array.
- O primeiro parâmetro `$data` para `HtmlHelper::style()` agora sempre irá exigir um array.

¹⁹ <http://getbootstrap.com/>

²⁰ <http://foundation.zurb.com/>

- O parâmetro `inline` foi removido dos métodos `meta()`, `css()`, `script()` e `scriptBlock()`. Ao invés disso, você deve usar a opção `block`. Definindo `block => true` irá emular o comportamento anterior.
- O `HtmlHelper::meta()` agora exige que o `$type` seja uma string. Opções adicionais podem ser passadas como `$options`.
- O `HtmlHelper::nestedList()` agora exige que o `$options` seja um array. O quarto argumento para o tipo `tag` foi removido e incluído no array `$options`.
- O argumento `$confirmMessage` de `Cake\View\Helper\HtmlHelper::link()` foi removido. Você deve usar agora a chave `confirm` no `$options` para especificar a mensagem.

PaginatorHelper

- O `link()` foi removido. Ele não era mais usado internamente pelo ajudante. Ele era pouco usado em códigos de usuários e não se encaixava mais nos objetivos do ajudante.
- O `next()` não tem mais as opções `'class'` ou `'tag'`. Ele não tem mais argumentos desabilitados. Ao invés disso são usados templates.
- O `prev()` não tem mais as opções `'class'` ou `'tag'`. Ele não tem mais argumentos desabilitados. Ao invés disso são usados templates.
- O `first()` não tem mais as opções `'after'`, `'ellipsis'`, `'separator'`, `'class'` ou `'tag'`.
- O `last()` não tem mais as opções `'after'`, `'ellipsis'`, `'separator'`, `'class'` ou `'tag'`.
- O `numbers()` não tem mais as opções `'separator'`, `'tag'`, `'currentTag'`, `'currentClass'`, `'class'`, `'tag'` e `'ellipsis'`. Essas opções são agora facilitadas pelos templates. Ele também exige que agora o parâmetro `$options` seja um array.
- O espaço reservado de estilo `%page%` foi removido de `Cake\View\Helper\PaginatorHelper::counter()`. Use o espaço reservado de estilo `{{page}}` no lugar.
- O `url()` foi renomeada para `generateUrl()` para evitar colisão de declaração de método com `Helper::url()`.

Por padrão todos os links e textos inativos são encapsulados em elementos ``. Isso ajuda a fazer o CSS mais fácil de escrever, e aumenta a compatibilidade com frameworks de CSS populares.

Ao invés de várias opções em cada método, você deve usar a funcionalidade de templates. Veja a documentação [PaginatorHelper Templates](#) para informações de como se usar templates.

TimeHelper

- `TimeHelper::__set()`, `TimeHelper::__get()`, e `TimeHelper::__isset()` foram removidos. Eles eram métodos mágicos para atributos obsoletos.
- O `TimeHelper::serverOffset()` foi removido. Ele provia práticas incorretas de operações com tempo.
- O `TimeHelper::niceShort()` foi removido.

NumberHelper

- O `NumberHelper::format()` agora exige que `$options` seja um array.

SessionHelper

- O `SessionHelper` está obsoleto. Você pode usar `$this->request->session()` diretamente, e a funcionalidade de mensagens flash foi movida para *Flash*.

JsHelper

- O `JsHelper` e todos motores associados foram removidos. Ele podia gerar somente um subconjunto muito pequeno de códigos JavaScript para biblioteca selecionada e consequentemente tentar gerar todo código JavaScript usando apenas o ajudante se tornava um impedimento com frequência. É recomendado usar diretamente sua biblioteca JavaScript preferida.

CacheHelper Removido

O `CacheHelper` foi removido. A funcionalidade de cache que ele fornecia não era padrão, limitada e incompatível com layouts não-HTML e views de dados. Essas limitações significavam que uma reconstrução completa era necessária. O ESI (Edge Side Includes) se tornou uma maneira padronizada para implementar a funcionalidade que o `CacheHelper` costumava fornecer. Entretanto, implementando [Edge Side Includes](http://en.wikipedia.org/wiki/Edge_Side_Includes)²¹ em PHP tem várias limitações e casos. Ao invés de construir uma solução ruim, é recomendado que os desenvolvedores que precisem de cache de resposta completa use o [Varnish](http://varnish-cache.org)²² ou [Squid](http://squid-cache.org)²³ no lugar.

I18n

O subsistema de internacionalização foi completamente reescrito. Em geral, você pode esperar o mesmo comportamento que nas versões anteriores, especialmente se você está usando a família de funções `__()`.

Internamente, a classe `I18n` usa `Aura\Intl`, e métodos apropriados são expostos para dar acesso a funções específicas da biblioteca. Por esta razão a maior parte dos métodos dentro de `I18n` foram removidos ou renomeados.

Devido ao uso do `ext/intl`, a classe `L10n` foi removida completamente. Ela fornecia dados incompletos e desatualizados em comparação com os dados disponíveis na classe `Locale` do PHP.

O idioma padrão da aplicação não será mais alterado automaticamente pelos idiomas aceitos pelo navegador nem por ter o valor `Config.language` definido na sessão do navegador. Você pode, entretanto, usar um filtro no despachante para trocar o idioma automaticamente a partir do cabeçalho `Accept-Language` enviado pelo navegador:

```
// No config/bootstrap.php
DispatcherFactory::addFilter('LocaleSelector');
```

Não há nenhum substituto incluso para selecionar automaticamente o idioma a partir de um valor configurado na sessão do usuário.

A função padrão para formatação de mensagens traduzidas não é mais a `sprintf`, mas a mais avançada e funcional classe `MessageFormatter`. Em geral você pode reescrever os espaços reservados nas mensagens como segue:

```
// Antes:
__('Hoje é um dia %s na %s', 'Ensolarado', 'Espanha');

// Depois:
__('Hoje é um dia {0} na {1}', 'Ensolarado', 'Espanha');
```

²¹ http://en.wikipedia.org/wiki/Edge_Side_Includes

²² <http://varnish-cache.org>

²³ <http://squid-cache.org>

Você pode evitar ter de reescrever suas mensagens usando o antigo formatador `sprintf`:

```
I18n::defaultFormatter('sprintf');
```

Adicionalmente, o valor `Config.language` foi removido e ele não pode mais ser usado para controlar o idioma atual da aplicação. Ao invés disso, você pode usar a classe `I18n`:

```
// Antes
Configure::write('Config.language', 'fr_FR');

// Agora
I18n::locale('en_US');
```

- Os métodos abaixo foram movidos:
 - De `Cake\I18n\Multibyte::utf8()` para `Cake\Utility\Text::utf8()`
 - De `Cake\I18n\Multibyte::ascii()` para `Cake\Utility\Text::ascii()`
 - De `Cake\I18n\Multibyte::checkMultibyte()` para `Cake\Utility\Text::isMultibyte()`
- Como agora o CakePHP requer a extensão `mbstring`, a classe `Multibyte` foi removida.
- As mensagens de erro por todo o CakePHP não passam mais através das funções de internacionalização. Isso foi feito para simplificar o núcleo do CakePHP e reduzir a sobrecarga. As mensagens apresentadas aos desenvolvedores são raramente, isso quando, são de fato traduzidas - de modo que essa sobrecarga adicional trás pouco benefício.

Localização

- Agora o construtor de `Cake\I18n\L10n` recebe uma instância de `Cake\Network\Request` como argumento.

Testes

- O `TestShell` foi removido. O CakePHP, o esqueleto da aplicação e novos plugins “cozinhados”, todos usam o `phpunit` para rodar os testes.
- O `webrunner` (`webroot/test.php`) foi removido. A adoção do CLI aumentou grandemente desde o release inicial do 2.x. Adicionalmente, os CLI de execução oferecem integração superior com IDE’s e outras ferramentas automáticas.

Se você sentir necessidade de um jeito de executar os testes a partir de um navegador, você deve verificar o [VisualPHPUnit](#)²⁴. Ele oferece muitas funcionalidades adicionais que o antigo `webrunner`.

- O `ControllerTestCase` está obsoleto e será removido no CakePHP 3.0.0. Ao invés disso, você deve usar a nova funcionalidade *Controller Integration Testing*.
- As `Fixtures` devem agora ser referenciadas usando sua forma no plural:

```
// No lugar de
$fixtures = ['app.artigo'];

// Você deve usar
$fixtures = ['app.artigos'];
```

²⁴ <https://github.com/NSinopoli/VisualPHPUnit>

Utilitários

Classe Set Removida

A classe Set foi removida, agora você deve usar a classe Hash no lugar dela.

Pastas & Arquivos

As classes de pastas e arquivos foram renomeadas:

- O `Cake\Utility\File` foi renomeado para `Cake\Filesystem\File`
- O `Cake\Utility\Folder` foi renomeado para `Cake\Filesystem\Folder`

Inflexão

- O valor padrão para o argumento `$replacement` do `Cake\Utility\Inflector::slug()` foi alterado do sublinhado (`_`) para o traço (`-`). Usando traços para separar palavras nas URLs é a escolha popular e também recomendada pelo Google.
- As transliterações para `Cake\Utility\Inflector::slug()` foram alteradas. Se você usa transliterações personalizadas você terá que atualizar seu código. No lugar de expressões regulares, as transliterações usam simples substituições de string. Isso rendeu melhorias de performance significativas:

```
// No lugar de
Inflector::rules('transliteration', [
    '/ä|æ/' => 'ae',
    '/å/' => 'aa'
]);

// Você deve usar
Inflector::rules('transliteration', [
    'ä' => 'ae',
    'æ' => 'ae',
    'å' => 'aa'
]);
```

- Os conjuntos distintos de regras de não-inflexões e irregulares para pluralização e singularização foram removidos. No lugar agora temos uma lista comum para cada. Quando usar `Cake\Utility\Inflector::rules()` com o tipo `'singular'` e `'plural'` você não poderá mais usar chaves como `'uninflected'` e `'irregular'` no array de argumentos `$rules`.

Você pode adicionar / sobrescrever a lista de regras de não-inflexionados e irregulares usando `Cake\Utility\Inflector::rules()` com valores `'uninflected'` e `'irregular'` para o argumento `$type`.

Sanitize

- A classe Sanitize foi removida.

Segurança

- O `Security::cipher()` foi removido. Ele era inseguro e promovia práticas ruins de criptografia. Você deve usar o `Security::encrypt()` no lugar.

- O valor de configuração `Security.cipherSeed` não é mais necessário. Com a remoção de `Security::cipher()` ele não tem utilidade.
- A retrocompatibilidade do `Cake\Utility\Security::rijndael()` para valores encriptados antes do CakePHP 2.3.1 foi removido. Você deve reencriptar os valores usando `Security::encrypt()` e uma versão recente do CakePHP 2.x antes de migrar.
- A habilidade para gerar um hash do tipo blowfish foi removido. Você não pode mais usar o tipo “blowfish” em `Security::hash()`. Deve ser usado apenas o `password_hash()` do PHP e `password_verify()` para gerar e verificar hashes blowfish. A compabilidade da biblioteca [ircmaxell/password-compat](https://packagist.org/packages/ircmaxell/password-compat)²⁵ que é instalado junto com o CakePHP fornece essas funções para versões de PHP menor que 5.5.
- O OpenSSL é usado agora no lugar do mcrypt ao encriptar/desenciptar dados. Essa alteração fornece uma melhor performance e deixa o CakePHP a prova de futuros abandonos de suporte das distribuições ao mcrypt.
- O `Security::rijndael()` está obsoleto e apenas disponível quando se usa o mcrypt.

Aviso: Dados encriptados com `Security::encrypt()` em versões anteriores não são compatíveis com a implementação openssl. Você deve *definir a implementação como mcrypt* quando fizer atualização.

Data e Hora

- O `CakeTime` foi renomeado para `Cake\I18n\Time`.
- O `CakeTime::serverOffset()` foi removido. Ele provia práticas incorretas de operações com tempo.
- O `CakeTime::niceShort()` foi removido.
- O `CakeTime::convert()` foi removido.
- O `CakeTime::convertSpecifiers()` foi removido.
- O `CakeTime::dayAsSql()` foi removido.
- O `CakeTime::daysAsSql()` foi removido.
- O `CakeTime::fromString()` foi removido.
- O `CakeTime::gmt()` foi removido.
- O `CakeTime::toATOM()` foi renomeado para `toAtomString`.
- O `CakeTime::toRSS()` foi renomeado para `toRssString`.
- O `CakeTime::toUnix()` foi renomeado para `toUnixString`.
- O `CakeTime::wasYesterday()` foi renomeado para `isYesterday` para combinar com o resto da renomeação de métodos.
- O `CakeTime::format()` não usa mais o formato do `sprintf`, ao invés disso você deve usar o formato `i18nFormat`.
- O `Time::timeAgoInWords()` agora exige que o `$options` seja um array.

A classe `Time` não é mais uma coleção de métodos estáticos, ela estende o `DateTime` para herdar todos seus métodos e adicionar funções de formatação baseado em localização com ajuda da extensão `intl`.

Em geral, expressões assim:

²⁵ <https://packagist.org/packages/ircmaxell/password-compat>


```
CakeTime::aMethod($date);
```

Podem ser migradas reescrevendo para:

```
(new Time($date))->aMethod();
```

Números

A biblioteca Number foi reescrita para usar internamente a classe NumberFormatter.

- O CakeNumber foi renomeada para Cake\I18n\Number.
- O Number::format() agora exige que o \$options seja um array.
- O Number::addFormat() foi removido.
- O Number::fromReadableSize() foi movido para Cake\Utility\Text::parseFileSize().

Validação

- A faixa de valores para Validation::range() agora é inclusiva se \$lower e \$upper forem fornecidos.
- O Validation::ssn() foi removido.

Xml

- O Xml::build() agora exige que o \$options seja um array.
- O Xml::build() não aceita mais uma URL. Se você precisar criar um documento XML a partir de uma URL, use [Http\Client](#).

Tutoriais & Exemplos

Nesta seção, você poderá caminhar através de típicas aplicações CakePHP para ver como todas as peças se encaixam. Como alternativa, você pode preferir visitar o repositório não oficial de plugins para o CakePHP [CakePackages](https://plugins.cakephp.org/)²⁶ e a [Bakery \(Padaria\)](https://bakery.cakephp.org/)²⁷ para conhecer aplicações e componentes existentes.

Tutorial - Criando um Bookmarker - Parte 1

Esse tutorial vai guiar você através da criação de uma simples aplicação de marcação (bookmarker). Para começar, nós vamos instalar o CakePHP, criar nosso banco de dados, e usar as ferramentas que o CakePHP fornece para obter nossa aplicação de pé rápido.

Aqui está o que você vai precisar:

1. Um servidor de banco de dados. Nós vamos usar o servidor MySQL neste tutorial. Você precisa saber o suficiente sobre SQL para criar um banco de dados: O CakePHP vai tomar as rédeas a partir daí. Por nós estarmos usando o MySQL, também certifique-se que você tem a extensão `pdo_mysql` habilitada no PHP.
2. Conhecimento básico sobre PHP.

Vamos começar!

Instalação do CakePHP

A maneira mais fácil de instalar o CakePHP é usando Composer, um gerenciador de dependências para o PHP. É uma forma simples de instalar o CakePHP a partir de seu terminal ou prompt de comando. Primeiro, você precisa baixar e instalar o Composer. Se você tiver instalada a extensão cURL do PHP, execute o seguinte comando:

```
curl -s https://getcomposer.org/installer | php
```

Ao invés disso, você também pode baixar o arquivo `composer.phar` do [site](https://getcomposer.org/)²⁸ oficial.

Em seguida, basta digitar a seguinte linha no seu terminal a partir do diretório onde se localiza o arquivo `composer.phar` para instalar o esqueleto de aplicações do CakePHP no diretório `bookmarker`.

²⁶ <https://plugins.cakephp.org/>

²⁷ <https://bakery.cakephp.org/>

²⁸ <https://getcomposer.org/download/>

```
php composer.phar create-project --prefer-dist cakephp/app bookmarker
```

A vantagem de usar Composer é que ele irá completar automaticamente um conjunto importante de tarefas, como configurar as permissões de arquivo e criar a sua **config/app.php**.

Há outras maneiras de instalar o CakePHP. Se você não puder ou não quiser usar Composer, veja a seção [Instalação](#).

Independentemente de como você baixou o CakePHP, uma vez que sua instalação for concluída, a estrutura dos diretórios deve ficar parecida com o seguinte:

```
/bookmarker
/bin
/config
/logs
/plugins
/src
/tests
/tmp
/vendor
/webroot
.editorconfig
.gitignore
.htaccess
.travis.yml
composer.json
index.php
phpunit.xml.dist
README.md
```

Agora pode ser um bom momento para aprender sobre como a estrutura de diretórios do CakePHP funciona: Confira a seção [Estrutura de pastas do CakePHP](#).

Verificando nossa instalação

Podemos checar rapidamente que a nossa instalação está correta, verificando a página inicial padrão. Antes que você possa fazer isso, você vai precisar iniciar o servidor de desenvolvimento:

```
bin/cake server
```

Isto irá iniciar o servidor embutido do PHP na porta 8765. Abra <http://localhost:8765> em seu navegador para ver a página de boas-vindas. Todas as verificações devem estar cheçadas corretamente, a não ser a conexão com banco de dados do CakePHP. Se não, você pode precisar instalar extensões do PHP adicionais, ou definir permissões de diretório.

Criando o banco de dados

Em seguida, vamos criar o banco de dados para a nossa aplicação. Se você ainda não tiver feito isso, crie um banco de dados vazio para uso nesse tutorial, com um nome de sua escolha, por exemplo, `cake_bookmarks`. Você pode executar o seguinte SQL para criar as tabelas necessárias:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
```

```

        modified DATETIME
    );

CREATE TABLE bookmarks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(50),
    description TEXT,
    url TEXT,
    created DATETIME,
    modified DATETIME,
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
);

CREATE TABLE bookmarks_tags (
    bookmark_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (bookmark_id, tag_id),
    INDEX tag_idx (tag_id, bookmark_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY bookmark_key(bookmark_id) REFERENCES bookmarks(id)
);

```

Você deve ter notado que a tabela `bookmarks_tags` utilizada uma chave primária composta. O CakePHP suporta chaves primárias compostas quase todos os lugares, tornando mais fácil construir aplicações multi-arrendados.

Os nomes de tabelas e colunas que usamos não foram arbitrários. Usando *convenções de nomenclatura* do CakePHP, podemos alavancar o desenvolvimento e evitar ter de configurar o framework. O CakePHP é flexível o suficiente para acomodar até mesmo esquemas de banco de dados legados inconsistentes, mas aderir às convenções vai lhe poupar tempo.

Configurando o banco de dados

Em seguida, vamos dizer ao CakePHP onde o nosso banco de dados está como se conectar a ele. Para muitos, esta será a primeira e última vez que você vai precisar configurar qualquer coisa.

A configuração é bem simples: basta alterar os valores do array `Datasources.default` no arquivo **config/app.php** pelos que se aplicam à sua configuração. A amostra completa da gama de configurações pode ser algo como o seguinte:

```

return [
    // Mais configuração acima.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',

```

```
'password' => 'AngelF00dC4k3~',
'database' => 'cake_bookmarks',
'encoding' => 'utf8',
'timezone' => 'UTC',
'cacheMetadata' => true,
    ],
    ],
    // Mais configuração abaixo.
];
```

Depois de salvar o seu arquivo **config/app.php**, você deve notar que a mensagem 'CakePHP is able to connect to the database' tem uma marca de verificação.

Nota: Uma cópia do arquivo de configuração padrão do CakePHP é encontrado em **config/app.default.php**.

Gerando o código base

Devido a nosso banco de dados seguir as convenções do CakePHP, podemos usar o *bake console* para gerar rapidamente uma aplicação básica. Em sua linha de comando execute:

```
bin/cake bake all users
bin/cake bake all bookmarks
bin/cake bake all tags
```

Isso irá gerar os controllers, models, views, seus casos de teste correspondentes, e fixtures para os nossos users, bookmarks e tags. Se você parou seu servidor, reinicie-o e vá para `http://localhost:8765/bookmarks`.

Você deverá ver uma aplicação que dá acesso básico, mas funcional a tabelas de banco de dados. Adicione alguns users, bookmarks e tags.

Adicionando criptografia de senha

Quando você criou seus users, você deve ter notado que as senhas foram armazenadas como texto simples. Isso é muito ruim do ponto de vista da segurança, por isso vamos consertar isso.

Este também é um bom momento para falar sobre a camada de modelo. No CakePHP, separamos os métodos que operam em uma coleção de objetos, e um único objeto em diferentes classes. Métodos que operam na recolha de entidades são colocadas na classe *Table*, enquanto as características pertencentes a um único registro são colocados na classe *Entity*.

Por exemplo, a criptografia de senha é feita no registro individual, por isso vamos implementar esse comportamento no objeto entidade. Dada a circunstância de nós querermos criptografar a senha cada vez que é definida, vamos usar um método modificador/definidor. O CakePHP vai chamar métodos de definição baseados em convenções a qualquer momento que uma propriedade é definida em uma de suas entidades. Vamos adicionar um definidor para a senha. Em **src/Model/Entity/User.php** adicione o seguinte:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Auth\DefaultPasswordHasher;

class User extends Entity
{
```

```
// Code from bake.

protected function _setPassword($value)
{
    $hasher = new DefaultPasswordHasher();
    return $hasher->hash($value);
}
}
```

Agora atualize um dos usuários que você criou anteriormente, se você alterar sua senha, você deve ver um senha criptografada ao invés do valor original nas páginas de lista ou visualização. O CakePHP criptografa senhas com `bcrypt`²⁹ por padrão. Você também pode usar `sha1` ou `md5` caso venha a trabalhar com um banco de dados existente.

Recuperando bookmarks com uma tag específica

Agora que estamos armazenando senhas com segurança, podemos construir algumas características mais interessantes em nossa aplicação. Uma vez que você acumulou uma coleção de bookmarks, é útil ser capaz de pesquisar através deles por tag. Em seguida, vamos implementar uma rota, a ação do controller, e um método localizador para pesquisar através de bookmarks por tag.

Idealmente, nós teríamos uma URL que se parece com `http://localhost:8765/bookmarks/tagged/funny/cat/gifs`. Isso deveria nos permitir a encontrar todos os bookmarks que têm as tags ‘funny’, ‘cat’ e ‘gifs’. Antes de podermos implementar isso, vamos adicionar uma nova rota. Em `config/routes.php`, adicione o seguinte na parte superior do arquivo:

```
Router::scope(
    '/bookmarks',
    ['controller' => 'Bookmarks'],
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);
```

O acima define uma nova “rota” que liga o caminho `/bookmarks/tagged/*`, a `BookmarksController::tags()`. Ao definir rotas, você pode isolar como suas URLs parecerão, de como eles são implementadas. Se fôssemos visitar `http://localhost:8765/bookmarks/tagged`, deveríamos ver uma página de erro informativa do CakePHP. Vamos implementar esse método ausente agora. Em `src/Controller/BookmarksController.php` adicione o seguinte:

```
public function tags()
{
    $tags = $this->request->getParam('pass');
    $bookmarks = $this->Bookmarks->find('tagged', [
        'tags' => $tags
    ]);
    $this->set(compact('bookmarks', 'tags'));
}
```

Criando o método localizador

No CakePHP nós gostamos de manter as nossas ações do controller enxutas, e colocar a maior parte da lógica de nossa aplicação nos modelos. Se você fosse visitar a URL `/bookmarks/tagged` agora, você

²⁹ <http://codahale.com/how-to-safely-store-a-password/>

veria um erro sobre o método `findTagged` não estar implementado ainda, então vamos fazer isso. Em `src/Model/Table/BookmarksTable.php` adicione o seguinte:

```
public function findTagged(Query $query, array $options)
{
    $bookmarks = $this->find()
        ->select(['id', 'url', 'title', 'description']);

    if (empty($options['tags'])) {
        $bookmarks
            ->leftJoinWith('Tags')
            ->where(['Tags.title IS' => null]);
    } else {
        $bookmarks
            ->innerJoinWith('Tags')
            ->where(['Tags.title IN ' => $options['tags']]);
    }

    return $bookmarks->group(['Bookmarks.id']);
}
```

Nós implementamos um método *localizador customizado*. Este é um conceito muito poderoso no CakePHP que lhe permite construir consultas reutilizáveis. Em nossa pesquisa, nós alavancamos o método `matching()` que nos habilita encontrar bookmarks que têm uma tag ‘correspondente’.

Criando a view

Agora, se você visitar a URL `/bookmarks/tagged`, o CakePHP irá mostrar um erro e deixá-lo saber que você ainda não fez um arquivo view. Em seguida, vamos construir o arquivo view para a nossa ação `tags`. Em `src/Template/Bookmarks/tags.ctp` coloque o seguinte conteúdo:

```
<h1>
    Bookmarks tagged with
    <?= $this->Text->toList(h($tags)) ?>
</h1>

<section>
<?php foreach ($bookmarks as $bookmark): ?>
    <article>
        <h4><?= $this->Html->link($bookmark->title, $bookmark->url) ?></h4>
        <small><?= h($bookmark->url) ?></small>
        <?= $this->Text->autoParagraph(h($bookmark->description)) ?>
    </article>
<?php endforeach; ?>
</section>
```

O CakePHP espera que os nossos templates sigam a convenção de nomenclatura onde o nome do template é a versão minúscula e grifada do nome da ação do controller.

Você pode perceber que fomos capazes de utilizar as variáveis `$tags` e `bookmarks` em nossa view. Quando usamos o método `set()` em nosso controller, automaticamente definimos variáveis específicas que devem ser enviadas para a view. A view vai tornar todas as variáveis passadas disponíveis nos templates como variáveis locais.

Em nossa view, usamos alguns dos *helpers* nativos do CakePHP. Helpers são usados para criar lógica re-utilizável para a formatação de dados, a criação de HTML ou outra saída da view.

Agora você deve ser capaz de visitar a URL `/bookmarks/tagged/funny` e ver todas os bookmarks com a tag ‘funny’.

Até agora, nós criamos uma aplicação básica para gerenciar bookmarks, tags e users. No entanto, todos podem ver as tags de toda a gente. No próximo capítulo, vamos implementar a autenticação e restringir os bookmarks visíveis para somente aqueles que pertencem ao usuário atual.

Agora vá a [Tutorial - Criando um Bookmarker - Parte 2](#) para continuar a construir sua aplicação ou mergulhe na documentação para saber mais sobre o que CakePHP pode fazer por você.

Tutorial - Criando um Bookmarker - Parte 2

Depois de terminar a *primeira parte deste tutorial*, você deve ter uma aplicação muito básica. Neste capítulo iremos adicionar autenticação e restringir as bookmarks para que cada usuário possa ver/modificar somente aquelas que possam.

Adicionando login

No CakePHP, a autenticação é feita por *Components (Componentes)*. Os Components podem ser considerados como formas de criar pedaços reutilizáveis de código relacionado a controllers com uma característica específica ou conceito. Os components também podem ligar-se ao evento do ciclo de vida do controller e interagir com a sua aplicação. Para começar, vamos adicionar o AuthComponent a nossa aplicação. Nós vamos querer muito que cada método exija autenticação, por isso vamos acrescentar o *AuthComponent* em nosso ApplicationController:

```
// Em src/Controller/AppController.php
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'authenticate' => [
                'Form' => [
                    'fields' => [
                        'username' => 'email',
                        'password' => 'password'
                    ]
                ]
            ],
            'loginAction' => [
                'controller' => 'Users',
                'action' => 'login'
            ]
        ]);

        // Permite a ação display, assim nosso pages controller
        // continua a funcionar.
        $this->Auth->allow(['display']);
    }
}
```

Acabamos de dizer ao CakePHP que queremos carregar os components Flash e Auth. Além disso, temos a configuração personalizada do AuthComponent, assim a nossa tabela users pode usar email como username. Agora, se

se você for para qualquer URL, você vai ser chutado para `/users/login`, que irá mostrar uma página de erro já que não escrevemos o código ainda. Então, vamos criar a ação de login:

```
// Em src/Controller/UsersController.php

public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error('Your username or password is incorrect.');
```

E em `src/Template/Users/login.ctp` adicione o seguinte:

```
<h1>Login</h1>
<?= $this->Form->create() ?>
<?= $this->Form->input('email') ?>
<?= $this->Form->input('password') ?>
<?= $this->Form->button('Login') ?>
<?= $this->Form->end() ?>
```

Agora que temos um simples formulário de login, devemos ser capazes de efetuar login com um dos users que tenham senha criptografada.

Nota: Se nenhum de seus users tem senha criptografada, comente a linha `loadComponent('Auth')`. Então vá e edite o user, salvando uma nova senha para ele.

Agora você deve ser capaz de entrar. Se não, certifique-se que você está usando um user que tenha senha criptografada.

Adicionando logout

Agora que as pessoas podem efetuar o login, você provavelmente vai querer fornecer uma maneira de encerrar a sessão também. Mais uma vez, no `UsersController`, adicione o seguinte código:

```
public function logout()
{
    $this->Flash->success('You are now logged out.');
```

Agora você pode visitar `/users/logout` para sair e ser enviado à página de login.

Ativando inscrições

Se você não estiver logado e tentar visitar `/usuários` / adicionar você vai ser expulso para a página de login. Devemos corrigir isso se quisermos que as pessoas se inscrevam em nossa aplicação. No `UsersController` adicione o seguinte:

```
public function beforeFilter(\Cake\Event\Event $event)
{
```

```
$this->Auth->allow(['add']);
}
```

O texto acima diz ao AuthComponent que a ação add não requer autenticação ou autorização. Você pode querer dedicar algum tempo para limpar a /users/add e remover os links enganosos, ou continuar para a próxima seção. Nós não estaremos construindo a edição do usuário, visualização ou listagem neste tutorial, então eles não vão funcionar, já que o AuthComponent vai negar-lhe acesso a essas ações do controller.

Restringindo acesso

Agora que os usuários podem conectar-se, nós vamos querer limitar os bookmarks que podem ver para aqueles que fizeram. Nós vamos fazer isso usando um adaptador de 'autorização'. Sendo os nossos requisitos bastante simples, podemos escrever um código em nossa BookmarksController. Mas antes de fazer isso, vamos querer dizer ao AuthComponent como nossa aplicação vai autorizar ações. Em seu ApplicationController adicione o seguinte:

```
public function isAuthorized($user)
{
    return false;
}
```

Além disso, adicione o seguinte à configuração para Auth em seu ApplicationController:

```
'authorize' => 'Controller',
```

Seu método initialize agora deve parecer com:

```
public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize' => 'Controller', //added this line
        'authenticate' => [
            'Form' => [
                'fields' => [
                    'username' => 'email',
                    'password' => 'password'
                ]
            ]
        ],
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login'
        ],
        'unauthorizedRedirect' => $this->referer()
    ]);

    // Permite a ação display, assim nosso pages controller
    // continua a funcionar.
    $this->Auth->allow(['display']);
}
```

Vamos usar como padrão, negação do acesso, e de forma incremental conceder acesso onde faça sentido. Primeiro, vamos adicionar a lógica de autorização para os bookmarks. Em seu BookmarksController adicione o seguinte:

```
public function isAuthorized($user)
{
```

```

$action = $this->request->params['action'];

// As ações add e index são permitidas sempre.
if (in_array($action, ['index', 'add', 'tags'])) {
    return true;
}
// Todas as outras ações requerem um id.
if (!$this->request->getParam('pass.0')) {
    return false;
}

// Checa se o bookmark pertence ao user atual.
$id = $this->request->getParam('pass.0');
$bookmark = $this->Bookmarks->get($id);
if ($bookmark->user_id == $user['id']) {
    return true;
}
return parent::isAuthorized($user);
}

```

Agora, se você tentar visualizar, editar ou excluir um bookmark que não pertença a você, você deve ser redirecionado para a página de onde veio. No entanto, não há nenhuma mensagem de erro sendo exibida, então vamos corrigir isso a seguir:

```

// In src/Template/Layout/default.ctp
// Under the existing flash message.
<?= $this->Flash->render('auth') ?>

```

Agora você deve ver as mensagens de erro de autorização.

Corrigindo a view de listagem e formulários

Enquanto view e delete estão trabalhando, edit, add e index tem alguns problemas:

1. Ao adicionar um bookmark, você pode escolher o user.
2. Ao editar um bookmark, você pode escolher o user.
3. A página de listagem mostra os bookmarks de outros users.

Vamos enfrentar o formulário de adição em primeiro lugar. Para começar remova o input ('user_id') a partir de `src/Template/Bookmarks/add.ctp`. Com isso removido, nós também vamos atualizar o método add:

```

public function add()
{
    $bookmark = $this->Bookmarks->newEntity();
    if ($this->request->is('post')) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
        ↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
```

```
$this->set(compact('bookmark', 'tags'));
}
```

Ao definir a propriedade da entidade com os dados da sessão, nós removemos qualquer possibilidade do user modificar de que outro user um bookmark seja. Nós vamos fazer o mesmo para o formulário edit e action edit. Sua ação edit deve ficar assim:

```
public function edit($id = null)
{
    $bookmark = $this->Bookmarks->get($id, [
        'contain' => ['Tags']
    ]);
    if ($this->request->is(['patch', 'post', 'put'])) {
        $bookmark = $this->Bookmarks->patchEntity($bookmark, $this->request->
        ↪getData());
        $bookmark->user_id = $this->Auth->user('id');
        if ($this->Bookmarks->save($bookmark)) {
            $this->Flash->success('The bookmark has been saved.');
```

```
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error('The bookmark could not be saved. Please, try again.');
```

```
    }
    $tags = $this->Bookmarks->Tags->find('list');
    $this->set(compact('bookmark', 'tags'));
}
```

View de listagem

Agora, nós precisamos apenas exibir bookmarks para o user logado. Nós podemos fazer isso ao atualizar a chamada para `paginate()`. Altere sua ação index:

```
public function index()
{
    $this->paginate = [
        'conditions' => [
            'Bookmarks.user_id' => $this->Auth->user('id'),
        ]
    ];
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
}
```

Nós também devemos atualizar a ação `tags()` e o método localizador relacionado, mas vamos deixar isso como um exercício para que você conclua por si.

Melhorando a experiência com as tags

Agora, adicionar novas tags é um processo difícil, pois o `TagsController` proíbe todos os acessos. Em vez de permitir o acesso, podemos melhorar a interface do usuário para selecionar tags usando um campo de texto separado por vírgulas. Isso permitirá dar uma melhor experiência para os nossos usuários, e usar mais alguns grandes recursos no ORM.

Adicionando um campo computado

Porque nós queremos uma maneira simples de acessar as tags formatados para uma entidade, podemos adicionar um campo virtual/computado para a entidade. Em **src/Model/Entity/Bookmark.php** adicione o seguinte:

```
use Cake\Collection\Collection;

protected function _getTagString()
{
    if (isset($this->_properties['tag_string'])) {
        return $this->_properties['tag_string'];
    }
    if (empty($this->tags)) {
        return '';
    }
    $tags = new Collection($this->tags);
    $str = $tags->reduce(function ($string, $tag) {
        return $string . $tag->title . ', ';
    }, '');
    return trim($str, ', ');
}
```

Isso vai nos deixar acessar a propriedade computada `$bookmark->tag_string`. Vamos usar essa propriedade em inputs mais tarde. Lembre-se de adicionar a propriedade `tag_string` a lista `_accessible` em sua entidade.

Em **src/Model/Entity/Bookmark.php** adicione o `tag_string` ao `_accessible` desta forma:

```
protected $_accessible = [
    'user_id' => true,
    'title' => true,
    'description' => true,
    'url' => true,
    'user' => true,
    'tags' => true,
    'tag_string' => true,
];
```

Atualizando as views

Com a entidade atualizado, podemos adicionar uma nova entrada para as nossas tags. Nas views `add` e `edit`, substitua `tags._ids` pelo seguinte:

```
<?= $this->Form->input('tag_string', ['type' => 'text']) ?>
```

Persistindo a string tag

Agora que podemos ver as tags como uma string existente, vamos querer salvar os dados também. Por marcar o `tag_string` como acessível, o ORM irá copiar os dados do pedido em nossa entidade. Podemos usar um método `beforeSave` para analisar a cadeia tag e encontrar/construir as entidades relacionadas. Adicione o seguinte em **src/Model/Table/BookmarksTable.php**:

```
public function beforeSave($event, $entity, $options)
{
    if ($entity->tag_string) {
        $entity->tags = $this->_buildTags($entity->tag_string);
    }
}
```

```

    }
}

protected function _buildTags($tagString)
{
    $new = array_unique(array_map('trim', explode(',', $tagString)));
    $out = [];
    $query = $this->Tags->find()
        ->where(['Tags.title IN' => $new]);

    // Remove tags existentes da lista de novas tags.
    foreach ($query->extract('title') as $existing) {
        $index = array_search($existing, $new);
        if ($index !== false) {
            unset($new[$index]);
        }
    }
    // Adiciona tags existentes.
    foreach ($query as $tag) {
        $out[] = $tag;
    }
    // Adiciona novas tags.
    foreach ($new as $tag) {
        $out[] = $this->Tags->newEntity(['title' => $tag]);
    }
    return $out;
}

```

Embora esse código seja um pouco mais complicado do que o que temos feito até agora, ele ajuda a mostrar o quão poderosa a ORM do CakePHP é. Você pode facilmente manipular resultados da consulta usando os métodos de *Collections* (*Coleções*), e lidar com situações em que você está criando entidades sob demanda com facilidade.

Terminando

Nós expandimos nossa aplicação bookmarker para lidar com situações de autenticação e controle de autorização/acesso básico. Nós também adicionamos algumas melhorias agradáveis à UX, aproveitando os recursos FormHelper e ORM.

Obrigado por dispor do seu tempo para explorar o CakePHP. Em seguida, você pode saber mais sobre o *Models* (*Modelos*), ou você pode ler os /topics.

Tutorial - Criando um Blog - Parte 1

Este tutorial irá orientá-lo através da criação de um simples blog. Faremos a instalação do CakePHP, criaremos um banco de dados e implementaremos a lógica capaz de listar, adicionar, editar e apagar postagens do blog.

Aqui está o que você vai precisar:

1. Um servidor web em funcionamento. Nós iremos assumir que você esteja usando o Apache, embora as instruções para outros servidores sejam bem similares. Talvez seja preciso alterar um pouco a configuração do servidor, mas a maioria das pessoas pode ter o CakePHP instalado e funcionando sem qualquer trabalho extra. Certifique-se de que você tem o PHP 5.6.0 ou superior, e que as extensões *mbstring* e *intl* estejam habilitadas no PHP. Caso não saiba a versão do PHP que está instalada, utilize a função `phpinfo()` ou digite `php -v` no seu terminal de comando.

2. Um servidor de banco de dados. Nós vamos usar o servidor *MySQL* neste tutorial. Você precisa saber o mínimo sobre SQL para então criar um banco de dados, depois disso o CakePHP vai assumir as rédeas. Já que usaremos o *MySQL*, também certifique-se que a extensão `pdo_mysql` está habilitada no PHP.
3. Conhecimento básico sobre PHP.

Vamos começar!

Instalação do CakePHP

A maneira mais fácil de instalar o CakePHP é usando Composer, um gerenciador de dependências para o PHP. Se trata de uma forma simples de instalar o CakePHP a partir de seu terminal ou prompt de comando. Primeiro, você precisa baixar e instalar o Composer. Se possuir instalada a extensão *cURL* do PHP, execute o seguinte comando:

```
curl -s https://getcomposer.org/installer | php
```

Você também pode baixar o arquivo `composer.phar` do [site³⁰](https://getcomposer.org/) oficial do Composer.

Em seguida, basta digitar a seguinte linha de comando no seu terminal a partir do diretório onde se localiza o arquivo `composer.phar` para instalar o esqueleto da aplicação do CakePHP no diretório `[nome_do_app]`.

```
php composer.phar create-project --prefer-dist cakephp/app [nome_do_app]
```

A vantagem de usar o Composer é que ele irá completar automaticamente um conjunto importante de tarefas, como configurar corretamente as permissões de pastas e criar o **config/app.php** para você.

Há outras maneiras de instalar o CakePHP. Se você não puder ou não quiser usar o Composer, confira a seção *Instalação*.

Independentemente de como você baixou o CakePHP, uma vez que sua instalação for concluída, a estrutura dos diretórios deve ficar parecida com o seguinte:

```
/nome_do_app
/bin
/config
/logs
/plugins
/src
/tests
/tmp
/vendor
/webroot
.editorconfig
.gitignore
.htaccess
.travis.yml
composer.json
index.php
phpunit.xml.dist
README.md
```

Agora pode ser um bom momento para aprender sobre como a estrutura de diretórios do CakePHP funciona: Confira a seção *Estrutura de pastas do CakePHP*.

³⁰ <https://getcomposer.org/download/>

Permissões dos diretórios tmp e logs

Os diretórios **tmp** e **logs** precisam ter permissões adequadas para que possam ser alterados pelo seu servidor web. Se você usou o Composer na instalação, ele deve ter feito isso por você e confirmado com uma mensagem “Permissions set on <folder>”. Se você ao invés disso, recebeu uma mensagem de erro ou se quiser fazê-lo manualmente, a melhor forma seria descobrir por qual usuário o seu servidor web é executado (`<? = 'whoami' ; ?>`) e alterar o proprietário desses dois diretórios para este usuário. Os comandos finais a serem executados (em *nix) podem ser algo como:

```
chown -R www-data tmp
chown -R www-data logs
```

Se por alguma razão o CakePHP não puder escrever nesses diretórios, você será informado por uma advertência enquanto não estiver em modo de produção.

Embora não seja recomendado, se você é incapaz de redefinir as permissões do seu servidor web, você pode simplesmente alterar as permissões de gravação diretamente nos diretórios, executando os seguintes comandos:

```
chmod 777 -R tmp
chmod 777 -R logs
```

Criando o banco de dados do Blog

Em seguida, vamos configurar o banco de dados para o nosso blog. Se você ainda não tiver feito isto, crie um banco de dados vazio para usar neste tutorial, com um nome de sua escolha, por exemplo, `cake_blog`. Agora, vamos criar uma tabela para armazenar nossos artigos:

```
/* Primeiro, criamos a tabela articles: */
CREATE TABLE articles (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Nós vamos também inserir alguns artigos para usarmos em nossos testes. Execute os seguintes comandos SQL em seu banco de dados:

```
/* Então inserimos articles para testes: */
INSERT INTO articles (title,body,created)
VALUES ('The title', 'This is the article body.', NOW());
INSERT INTO articles (title,body,created)
VALUES ('A title once again', 'And the article body follows.', NOW());
INSERT INTO articles (title,body,created)
VALUES ('Title strikes back', 'This is really exciting! Not.', NOW());
```

Os nomes de tabelas e colunas que usamos não foram arbitrários. Usando *convenções de nomenclatura* do CakePHP, podemos alavancar o desenvolvimento e acelerar a configuração do framework. O CakePHP é flexível o suficiente para acomodar até mesmo esquemas de banco de dados legados inconsistentes, mas aderir às convenções vai lhe poupar tempo.

Configurando o banco de dados do Blog

Em seguida, vamos dizer ao CakePHP onde nosso banco de dados está e como se conectar a ele. Para muitos, esta será a primeira e última vez que será necessário configurar algo.

A configuração é bem simples e objetiva: basta alterar os valores no array `Datasources.default` localizado no arquivo **config/app.php**, pelos valores que se aplicam à sua configuração. Um exemplo completo de configurações deve se parecer como o seguinte:

```
return [
    // Mais configurações acima.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_blog',
            'encoding' => 'utf8',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // Mais configurações abaixo.
];
```

Depois de salvar o arquivo **config/app.php**, você deve notar a mensagem *CakePHP is able to connect to the database* ao acessar o Blog pelo seu navegador.

Nota: Uma cópia do arquivo de configuração padrão do CakePHP pode ser encontrada em **config/app.default.php**.

Configurações opcionais

Há alguns outros itens que podem ser configurados. Muitos desenvolvedores completam esta lista de itens, mas os mesmos não são obrigatórios para este tutorial. Um deles é definir uma sequência personalizada (ou “salt”) para uso em hashes de segurança.

A sequência personalizada (ou salt) é utilizada para gerar hashes de segurança. Se você utilizou o Composer, ele cuidou disso para você durante a instalação. Apesar disso, você precisa alterar a sequência personalizada padrão editando o arquivo **config/app.php**. Não importa qual será o novo valor, somente deverá ser algo difícil de descobrir:

```
'Security' => [
    'salt' => 'algum valor longo contendo uma mistura aleatória de valores.',
],
```

Observação sobre o mod_rewrite

Ocasionalmente, novos usuários irão se atrapalhar com problemas de mod_rewrite. Por exemplo, se a página de boas vindas do CakePHP parecer estranha (sem imagens ou estilos CSS). Isto provavelmente significa que o mod_rewrite não está funcionando em seu servidor. Por favor, verifique a seção *Reescrita de URL* para obter ajuda e resolver qualquer problema relacionado.

Agora continue o tutorial em *Tutorial - Criando um Blog - Parte 2* e inicie a construção do seu Blog com o CakePHP.

Tutorial - Criando um Blog - Parte 2

Criando o model

Após criar um model (modelo) no CakePHP, nós teremos a base necessária para interagirmos com o banco de dados e executar operações.

Os arquivos de classes, correspondentes aos models, no CakePHP estão divididos entre os objetos `Table` e `Entity`. Objetos `Table` provêm acesso à coleção de entidades armazenada em uma tabela e são alocados em **src/Model/Table**.

O arquivo que criaremos deverá ficar salvo em **src/Model/Table/ArticlesTable.php**:

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }
}
```

Convenções de nomenclatura são muito importantes no CakePHP. Ao nomear nosso objeto como `ArticlesTable`, o CakePHP automaticamente deduz que o mesmo utilize o `ArticlesController` e seja relacionado à tabela `articles`.

Nota: O CakePHP criará automaticamente um objeto model se não puder encontrar um arquivo correspondente em **src/Model/Table**. Se você nomear incorretamente seu arquivo (isto é, `articlestable.php` ou `ArticleTable.php`), o CakePHP não reconhecerá suas definições e usará o model gerado como alternativa.

Para mais informações sobre models, como callbacks e validação, visite o capítulo *Models (Modelos)* do manual.

Nota: Se você completou a *primeira parte* do tutorial e criou a tabela `articles`, você pode tomar proveito da capacidade de geração de código do `bake` através do console do CakePHP para criar o model `ArticlesTable`:

```
bin/cake bake model Articles
```

Para mais informações sobre o `bake` e suas características relacionadas a geração de código, visite o capítulo *Geração de código com bake* do manual.

Criando o controller

A seguir, criaremos um controller (controlador) para nossos artigos. O controller é responsável pela lógica de interação da aplicação. É o lugar onde você utilizará as regras contidas nos models e executará tarefas relacionadas aos artigos. Criaremos um arquivo chamado **ArticlesController.php** no diretório **src/Controller**:

```
// src/Controller/ArticlesController.php
```

```
namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Agora, vamos adicionar uma action (ação) ao nosso controller. Actions frequentemente, representam uma função ou interface em uma aplicação. Por exemplo, quando os usuários requisitarem `www.example.com/articles/index` (sendo o mesmo que `www.example.com/articles/`), eles esperam ver uma lista de artigos:

```
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
{
    public function index()
    {
        $articles = $this->Articles->find('all');
        $this->set(compact('articles'));
    }
}
```

Ao definir a função `index()` em nosso `ArticlesController`, os usuários podem acessá-la requisitando `www.example.com/articles/index`. Similarmente, se definíssemos uma função chamada `foobar()`, os usuários poderiam acessá-la em `www.example.com/articles/foobar`.

Aviso: Vocês podem ser tentados a nomear seus controllers e actions para obter uma certa URL. Resista a essa tentação. Siga as *Convenções do CakePHP* e crie nomes de action legíveis e compreensíveis. Você pode mapear URLs para o seu código utilizando *Roteamento*.

A instrução na action usa `set()` para passar dados do controller para a view. A variável é definida como `'articles'`, sendo igual ao valor retornado do método `find('all')` do objeto `ArticlesTable`.

Nota: Se você completou a *primeira parte* do tutorial e criou a tabela `articles`, você pode tomar proveito da capacidade de geração de código do bake através do console do CakePHP para criar o controller `ArticlesController`:

```
bin/cake bake controller Articles
```

Para mais informações sobre o bake e suas características sobre geração de código, visite o capítulo *Geração de código com bake* do manual.

Criando as views

Agora que nós temos os dados fluindo pelo nosso model, e nossa lógica da aplicação definida em nosso controller, vamos criar uma view (visualização) para a action `index()`.

As views do CakePHP são camadas de apresentação que se encaixam nos layouts da aplicação. Para a maioria das aplicações, elas são uma mescla entre HTML e PHP, mas também podem ser distribuídas como XML, CSV, ou ainda dados binários.

Um layout é um conjunto de códigos encontrado ao redor das views. Múltiplos layouts podem ser definidos, e você pode alterar entre eles, mas agora, vamos usar o default, localziado em **src/Template/Layout/default.ctp**.

Lembra que na última sessão atribuímos a variável ‘articles’ à view usando o método `set()`? Isso levará a coleção de objetos gerada pela query a ser invocada numa iteração `foreach`.

Arquivos de template do CakePHP são armazenados em **src/Template** dentro de uma pasta com o nome do controller correspondente (nós teremos que criar a pasta ‘Articles’ nesse caso). Para distribuir os dados de artigos em uma tabela, precisamos criar uma view assim:

```
<!-- File: src/Template/Articles/index.ctp -->

<h1>Blog articles</h1>
<table>
  <tr>
    <th>Id</th>
    <th>Title</th>
    <th>Created</th>
  </tr>

  <!-- Aqui é onde iremos iterar nosso objeto de solicitação $articles, exibindo_
  ↳ informações de artigos -->

  <?php foreach ($articles as $article): ?>
  <tr>
    <td><?= $article->id ?></td>
    <td>
      <?= $this->Html->link($article->title, ['action' => 'view', $article->
  ↳ id]) ?>
    </td>
    <td>
      <?= $article->created->format (DATE_RFC850) ?>
    </td>
  </tr>
  <?php endforeach; ?>
</table>
```

Você deve ter notado o uso de um objeto chamado `$this->Html`, uma instância da classe `Cake\View\Helper\HtmlHelper` do CakePHP. O CakePHP vem com um conjunto de view helpers que simplificam tarefas como gerar links e formulários. Você pode aprender como usá-los em *Helpers (Facilitadores)*, mas aqui é importante notar que o método `link()` irá gerar um link HTML com o referido título (primeiro parâmetro) e URL (segundo parâmetro).

Quando se especifica URLs no CakePHP, é recomendado o uso do formato de array. Isto será melhor explicado posteriormente na seção Rotas. Usando o formato de array para URLs, você toma vantagem das capacidades de roteamento reverso do CakePHP. Você também pode especificar URLs relativas a base da aplicação com o formato `/controller/action/param1/param2` ou usar *named routes*.

Neste ponto, você pode visitar <http://www.example.com/articles/index> no seu navegador. Você deve ver sua view corretamente formatada listando os artigos.

Se você clicar no link do título de um artigo listado, provavelmente será informado pelo CakePHP que a action ainda não foi definida, então vamos criá-la no `ArticlesController` agora:

```
// src/Controller/ArticlesController.php

namespace App\Controller;

class ArticlesController extends AppController
```

```
{

    public function index()
    {
        $this->set('articles', $this->Articles->find('all'));
    }

    public function view($id = null)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }
}
```

O uso do `set()` deve parecer familiar. Repare que você está usando `get()` ao invés de `find('all')` porque nós queremos a informação de apenas um artigo.

Repare que nossa action recebe um parâmetro: o ID do artigo que gostaríamos de visualizar. Esse parâmetro é entregue para a action através da URL solicitada. Se o usuário requisitar `/articles/view/3`, então o valor `'3'` é passado como `$id` para a action.

Ao usar a função `get()`, fazemos também algumas verificações para garantir que o usuário realmente está acessando um registro existente, se não ou se o `$id` for indefinido, a função irá lançar uma `NotFoundException`.

Agora vamos criar a view para nossa action em `src/Template/Articles/view.ctp`

```
<!-- File: src/Template/Articles/view.ctp -->

<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Criado: <?= $article->created->format (DATE_RFC850) ?></small></p>
```

Verifique se está tudo funcionando acessando os links em `/articles/index` ou manualmente solicite a visualização de um artigo acessando `articles/view/{id}`. Lembre-se de substituir `{id}` por um `'id'` de um artigo.

Adicionando artigos

Primeiro, comece criando a action `add()` no `ArticlesController`:

```
// src/Controller/ArticlesController.php

namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{

    public function initialize()
    {
        parent::initialize();

        $this->loadComponent('Flash'); // Inclui o FlashComponent
    }

    public function index()
    {
```

```

        $this->set('articles', $this->Articles->find('all'));
    }

    public function view($id)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }

    public function add()
    {
        $article = $this->Articles->newEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->
→getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Seu artigo foi salvo.'));
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Não é possível adicionar o seu artigo.'));
        }
        $this->set('article', $article);
    }
}

```

Nota: Você precisa incluir o *Flash* component em qualquer controller que vá usá-lo. Se necessário, inclua no `AppController` e assim o `FlashComponent` estará disponível para todos os controllers da aplicação.

A action `add()` checa se o método HTTP da solicitação foi POST, e então tenta salvar os dados utilizando o model `Articles`. Se por alguma razão ele não salvar, apenas renderiza a view. Isto nos dá a chance de exibir erros de validação ou outros alertas.

Cada requisição do CakePHP instancia um objeto `ServerRequest` que é acessível usando `$this->request`. O objeto contém informações úteis sobre a requisição que foi recebida e pode ser usado para controlar o fluxo de sua aplicação. Nesse caso, nós usamos o método `Cake\Network\ServerRequest::is()` para checar se a requisição é do tipo HTTP POST.

Quando se usa um formulário para postar dados, essa informação fica disponível em `$this->request->getData()`. Você pode usar as funções `pr()` ou `debug()` caso queira verificar esses dados.

Usamos os métodos `success()` e `error()` do `FlashComponent` para definir uma mensagem que será armazenada numa variável de sessão. Esses métodos são gerados usando os [recursos de métodos mágicos](#)³¹ do PHP. Mensagens flash serão exibidas na página após um redirecionamento. No layout nós temos `<?=$this->Flash->render() ?>` que exibe a mensagem e limpa a variável de sessão. A função do controller `Cake\Controller\Controller::redirect` redireciona para qualquer outra URL. O parâmetro `['action' => 'index']` corresponde a URL `/articles`, isto é, a action `index()` do `ArticlesController`. Você pode consultar a função `Cake\Routing\Router::url()` na API³² e checar os formatos a partir dos quais você pode montar uma URL.

Chamar o método `save()` vai checar erros de validação e abortar o processo caso os encontre. Nós vamos abordar como esses erros são tratados nas sessões a seguir.

³¹ <http://php.net/manual/en/language.oop5.overloading.php#object.call>

³² <https://api.cakephp.org>

Validando artigos

O CakePHP torna mais prática e menos monótona a validação de dados de formulário.

Para tirar proveito dos recursos de validação, você vai precisar usar o *Form* helper em suas views. O *Cake\View\Helper\FormHelper* está disponível por padrão em todas as views pelo uso do `$this->Form`.

Segue a view correspondente a action add:

```
<!-- File: src/Template/Articles/add.ctp -->

<h1>Add Article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->input('title');
    echo $this->Form->input('body', ['rows' => '3']);
    echo $this->Form->button(__('Salvar artigo'));
    echo $this->Form->end();
?>
```

Nós usamos o *FormHelper* para gerar a tag de abertura HTML de um formulário. Segue o HTML gerado por `$this->Form->create()`:

```
<form method="post" action="/articles/add">
```

Se `create()` é chamado sem parâmetros fornecidos, assume-se a construção de um formulário que submete dados via POST para a action `add()` (ou `edit()` no caso de um `id` estar incluído nos dados do formulário).

O método `$this->Form->input()` é usado para criar elementos do formulário do mesmo nome. O primeiro parâmetro diz ao CakePHP qual é o campo correspondente, e o segundo parâmetro permite que você especifique um vasto array de opções, nesse, o número de linhas para o textarea. `input()` vai gerar diferentes elementos de formulários baseados no tipo de campo especificado no model.

O `$this->Form->end()` fecha o formulário, entregando também elementos ocultos caso a prevenção contra CSRF/Form Tampering esteja habilitada.

Agora vamos voltar e atualizar nossa view **src/Template/Articles/index.ctp** para incluir um novo link. Antes do `<table>`, adicione a seguinte linha:

```
<?= $this->Html->link('Adicionar artigo', ['action' => 'add']) ?>
```

Você deve estar se perguntando: como eu digo ao CakePHP meus critérios de validação? Regras de validação são definidas no model. Vamos fazer alguns ajustes no nosso model:

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }

    public function validationDefault(Validator $validator)
```



```

{
    $validator
        ->notEmpty('title')
        ->notEmpty('body');

    return $validator;
}
}

```

O método `validationDefault()` diz ao CakePHP como validar seus dados quando o método `save()` for solicitado. Aqui, estamos especificando que tanto o campo `body` quanto `title` não devem estar vazios. O CakePHP possui muitos recursos de validação e um bom número de regras pré-determinadas (número de cartões, endereços de email, etc), além de flexibilidade para adicionar regras de validação customizadas. Para mais informações sobre configuração de validações, visite a documentação em [Validação](#).

Agora que suas regras de validação estão definidas, tente adicionar um artigo sem definir o campo `title` e `body` para ver como a validação funciona. Desde que tenhamos usado o método `Cake\View\Helper\FormHelper::input()` do `FormHelper` para criar nossos elementos, nossas mensagens de alerta da validação serão exibidas automaticamente.

Editando artigos

Edição, aí vamos nós! Você já é um profissional do CakePHP agora, então possivelmente detectou um padrão... Cria-se a action e então a view. Aqui segue a action `edit()` que deverá ser inserida no `ArticlesController`:

```

// src/Controller/ArticlesController.php

public function edit($id = null)
{
    $article = $this->Articles->get($id);
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Seu artigo foi atualizado.));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Seu artigo não pôde ser atualizado.));
    }

    $this->set('article', $article);
}

```

Essa action primeiramente certifica-se que o registro apontado existe. Se o parâmetro `$id` não foi passado ou se o registro é inexistente, uma `NotFoundException` é lançada pelo `ErrorHandler` do CakePHP.

Em seguida, a action verifica se a requisição é POST ou PUT e caso seja, os dados são usados para atualizar a entidade de artigo em questão ao usar o método `patchEntity()`. Então finalmente usamos o `ArticlesTable` para salvar a entidade.

Segue a view correspondente a action `edit`:

```

<!-- File: src/Template/Articles/edit.ctp -->

<h1>Edit Article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->input('title');

```

```
echo $this->Form->input('body', ['rows' => '3']);
echo $this->Form->button(__('Salvar artigo'));
echo $this->Form->end();
?>
```

Essa view retorna o formulário de edição com os dados populados, juntamente com qualquer mensagem de erro proveniente de validações.

O CakePHP irá determinar se o `save()` vai inserir ou atualizar um registro baseado nos dados da entidade.

Você pode atualizar sua view index com os links para editar artigos:

```
<!-- File: src/Template/Articles/index.ctp (edit links added) -->

<h1>Blog articles</h1>
<p><?= $this->Html->link("Adicionar artigo", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Título</th>
        <th>Criado</th>
        <th>Ações</th>
    </tr>

    <!-- Aqui é onde iremos iterar nosso objeto de solicitação $articles, exibindo_
    informações de artigos -->

    <?php foreach ($articles as $article): ?>
        <tr>
            <td><?= $article->id ?></td>
            <td>
                <?= $this->Html->link($article->title, ['action' => 'view', $article->
                id]) ?>
            </td>
            <td>
                <?= $article->created->format (DATE_RFC850) ?>
            </td>
            <td>
                <?= $this->Html->link('Editar', ['action' => 'edit', $article->id]) ?>
            </td>
        </tr>
    <?php endforeach; ?>
</table>
```

Deletando artigos

A seguir, vamos criar uma forma de deletar artigos. Comece com uma action `delete()` no `ArticlesController`:

```
// src/Controller/ArticlesController.php

public function delete($id)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->get($id);
```

```

if ($this->Articles->delete($article)) {
    $this->Flash->success(__('O artigo com id: {0} foi deletado.', h($id)));
    return $this->redirect(['action' => 'index']);
}
}

```

Essa lógica deleta o artigo especificado pelo \$id e usa \$this->Flash->success() para exibir uma mensagem de confirmação após o redirecionamento para /articles. Tentar excluir um registro usando uma requisição GET, fará com que o allowMethod() lance uma exceção. Exceções são capturadas pelo gerenciador de exceções do CakePHP e uma página de erro é exibida. Existem muitos *Exceptions* embutidos que podem indicar variados erros HTTP que sua aplicação possa precisar.

Por estarmos executando apenas lógica e redirecionando, essa action não tem uma view. Vamos atualizar nossa view index com links para excluir artigos:

```

<!-- File: src/Template/Articles/index.ctp (delete links added) -->

<h1>Blog articles</h1>
<p><?= $this->Html->link('Adicionar artigo', ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Título</th>
        <th>Criado</th>
        <th>Ações</th>
    </tr>

    <!-- Aqui é onde iremos iterar nosso objeto de solicitação $articles, exibindo_
    → informações de artigos -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->
            → id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Form->postLink(
                'Deletar',
                ['action' => 'delete', $article->id],
                ['confirm' => 'Tem certeza?'])
            ?>
            <?= $this->Html->link('Edit', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
    <?php endforeach; ?>
</table>

```

Usar View\Helper\FormHelper::postLink() vai criar um link que usa JavaScript para criar uma requisição POST afim de deletar um artigo.

Aviso: Permitir que registros sejam deletados usando requisições GET é perigoso, pois rastreadores na web podem acidentalmente deletar todo o seu conteúdo.

Nota: Esse código da view também usa o `FormHelper` para confirmar a action através de JavaScript.

Rotas

Para muitos o roteamento padrão do CakePHP funciona bem o suficiente. Desenvolvedores que consideram facilidade de uso e SEO irão apreciar a forma como o CakePHP mapeia determinadas URLs para actions específicas. Vamos realizar uma pequena mudança nas rotas neste tutorial.

Para mais informações sobre técnicas avançadas de roteamento, visite [Connecting Routes](#).

Por padrão, o CakePHP responde a uma requisição pela raiz do seu site usando o `PagesController`, ao renderizar uma view chamada **home.ctp**. Alternativamente, nós vamos substituir esse comportamento pelo `ArticlesController` ao criar uma regra de roteamento.

A configuração de rotas do CakePHP pode ser encontrada em **config/routes.php**. Você deve comentar ou remover a linha que define o roteamento padrão:

```
$routes->connect('/', ['controller' => 'Pages', 'action' => 'display', 'home']);
```

Essa linha conecta a URL `/` com a página padrão do CakePHP. Nós queremos que ela conecte-se ao nosso próprio controller, então a substitua por esta:

```
$routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Isso irá conectar requisições por `/` a action `index()` do nosso `ArticlesController`

Nota: O CakePHP aproveita-se do uso de roteamento reverso. Se com a rota anterior definida você gerar um link com a seguinte estrutura de array: `['controller' => 'Articles', 'action' => 'index']`, a URL resultante será `/`. Portanto, é uma boa ideia sempre usar arrays para URLs, pois assim suas rotas definem o endereço gerado e certificam-se que os links apontem sempre para o mesmo lugar.

Conclusão

Simples, não é? Tenha em mente que esse foi um tutorial básico. O CakePHP tem *muito* mais recursos a oferecer. Não abordamos outros tópicos aqui para manter a simplicidade. Use o restante do manual como um guia para criar aplicações mais ricas.

Agora que você criou uma aplicação básica no CakePHP, você pode continuar no [Tutorial - Criando um Blog - Parte 3](#), ou começar seu próprio projeto. Você também pode folhear os `/topics` ou a *API* <<https://api.cakephp.org/3.0>> para aprender mais sobre o CakePHP.

Se você precisar de ajuda, há muitas formas de conseguir, por favor, visite a página [Onde Conseguir Ajuda](#) e bem-vindo(a) ao CakePHP!

Leitura complementar

Existem tópicos comuns que as pessoas que estão estudando o CakePHP normalmente visitam a seguir:

1. *Layouts*: Customizando o layout da aplicação
2. *Elements*: Inclusão e reutilização de elementos na view
3. *Geração de código com bake*: Gerando código CRUD
4. ***Tutorial - Criando um Blog - Autenticação e Autorização***: Tutorial de autorização e autenticação

Tutorial - Criando um Blog - Parte 3

Criar uma árvore de Categoria

Vamos continuar o nosso aplicativo de blog e imaginar que queremos categorizar os nossos artigos. Queremos que as categorias sejam ordenadas, e para isso, vamos usar o comportamento de árvore para nos ajudar a organizar as categorias.

Mas primeiro, precisamos modificar nossas tabelas.

Migração de Plugin

Nós vamos usar o plugin de migrações para criar uma tabela em nosso banco de dados. Se você tem a tabela `articles` no seu banco de dados, apague. Agora abra o arquivo `composer.json` do seu aplicativo. Normalmente, você veria que o plugin de migração já está requisitando. Se não, adicione através da execução:

```
composer require cakephp/migrations:~1.0
```

O plugin de migração agora está na pasta de sua aplicação. Também, adicionar `Plugin::load('Migrations')` ; para o arquivo `bootstrap.php` do seu aplicativo.

Uma vez que o plugin está carregado, execute o seguinte comando para criar um arquivo de migração:

```
bin/cake bake migration CreateArticles title:string body:text category_id:integer_  
↳ created modified
```

Um arquivo de migração será gerado na pasta `/config/Migrations` com o seguinte:

```
<?php  
  
use Migrations\AbstractMigration;  
  
class CreateArticles extends AbstractMigration  
{  
    public function change()  
    {  
        $table = $this->table('articles');  
        $table->addColumn('title', 'string', [  
            'default' => null,  
            'limit' => 255,  
            'null' => false,  
        ]);  
        $table->addColumn('body', 'text', [  
            'default' => null,  
            'null' => false,  
        ]);  
        $table->addColumn('category_id', 'integer', [  
            'default' => null,
```

```
        'limit' => 11,
        'null' => false,
    ));
    $table->addColumn('created', 'datetime', [
        'default' => null,
        'null' => false,
    ]);
    $table->addColumn('modified', 'datetime', [
        'default' => null,
        'null' => false,
    ]);
    $table->create();
}
}
```

Executar outro comando para criar uma tabela de categorias. Se você precisar especificar um comprimento de campo, você pode fazê-lo dentro de colchetes no tipo de campo, ou seja:

```
bin/cake bake migration CreateCategories parent_id:integer lft:integer[10] right:
↳integer[10] name:string[100] description:string created modified
```

Isso irá gerar o seguinte arquivo no config/Migrations:

```
<?php

use Migrations\AbstractMigration;

class CreateCategories extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('categories');
        $table->addColumn('parent_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addColumn('lft', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('right', 'integer', [
            'default' => null,
            'limit' => 10,
            'null' => false,
        ]);
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 100,
            'null' => false,
        ]);
        $table->addColumn('description', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
```

```

        'default' => null,
        'null' => false,
    ));
    $table->addColumn('modified', 'datetime', [
        'default' => null,
        'null' => false,
    ]);
    $table->create();
    }
}

```

Agora que os arquivos de migração estão criados, você pode editá-los antes de criar suas tabelas. Precisamos mudar o 'null' => false para o campo parent_id com 'null' => true porque uma categoria de nível superior tem null no parent_id

Execute o seguinte comando para criar suas tabelas:

```
bin/cake migrations migrate
```

Modificando as Tabelas

Com nossas tabelas configuradas, agora podemos nos concentrar em categorizar os nossos artigos.

Supomos que você já tem os arquivos (Tabelas, controladores e modelos dos artigos) da parte 2. Então vamos adicionar as referências a categorias.

Precisamos associar os artigos e categorias juntos nas tabelas. Abra o arquivo src/Model/Table/ArticlesTable.php e adicione o seguinte:

```

// src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
        // Just add the belongsTo relation with CategoriesTable
        $this->belongsTo('Categories', [
            'foreignKey' => 'category_id',
        ]);
    }
}

```

Gerar código esqueleto por categorias

Crie todos os arquivos pelo comando bake:

```

bin/cake bake model Categories
bin/cake bake controller Categories
bin/cake bake template Categories

```

A ferramenta bake criou todos os seus arquivos em um piscar de olhos. Você pode fazer uma leitura rápida se quiser familiarizar como o CakePHP funciona.

Nota: Se você estiver no Windows lembre-se de usar \ em vez de /.

Você vai precisar editar o seguinte em `src/Template/Categories/add.ctp` e `src/Template/Categories/edit.ctp`:

```
echo $this->Form->input('parent_id', [
    'options' => $parentCategories,
    'empty' => 'No parent category'
]);
```

Anexar árvore de compartimento para CategoriesTable

O *TreeBehavior* ajuda você a gerenciar as estruturas de árvore hierárquica na tabela do banco de dados. Usa a lógica MPTT para gerenciar os dados. Estruturas de árvore MPTT são otimizados para lê, o que muitas vezes torna uma boa opção para aplicações pesadas, como ler blogs.

Se você abrir o arquivo `src/Model/Table/CategoriesTable.php`, você verá que o *TreeBehavior* foi anexado a sua *CategoriesTable* no método `initialize()`. Bake acrescenta esse comportamento para todas as tabelas que contêm `lft` e `right`:

```
$this->addBehavior('Tree');
```

Com o *TreeBehavior* anexado você vai ser capaz de acessar alguns recursos como a reordenação das categorias. Vamos ver isso em um momento.

Mas, por agora, você tem que remover as seguintes entradas em seus *Categorias* de adicionar e editar arquivos de modelo:

```
echo $this->Form->input('lft');
echo $this->Form->input('right');
```

Além disso, você deve desabilitar ou remover o `requirePresence` do validador, tanto para a `lft` e `right` nas colunas em seu modelo *CategoriesTable*:

```
public function validationDefault(Validator $validator)
{
    $validator
        ->add('id', 'valid', ['rule' => 'numeric'])
        ->allowEmpty('id', 'create');

    $validator
        ->add('lft', 'valid', ['rule' => 'numeric'])
        // ->requirePresence('lft', 'create')
        ->notEmpty('lft');

    $validator
        ->add('right', 'valid', ['rule' => 'numeric'])
        // ->requirePresence('right', 'create')
        ->notEmpty('right');
}
```

Esses campos são automaticamente gerenciados pelo *TreeBehavior* quando uma categoria é salvo.

Usando seu navegador, adicione algumas novas categorias usando os `/yoursite/categories/add` ação do controlador.

Reordenar categorias com TreeBehavior

Em seu arquivo de modelo de índices de categorias, você pode listar as categorias e reordená-los.

Vamos modificar o método de índice em sua `CategoriesController.php` e adicionar `moveUp()` e `moveDown()` para ser capaz de reordenar as categorias na árvore:

```
class CategoriesController extends AppController
{
    public function index()
    {
        $categories = $this->Categories->find()
            ->order(['lft' => 'ASC']);
        $this->set(compact('categories'));
        $this->set('_serialize', ['categories']);
    }

    public function moveUp($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveUp($category)) {
            $this->Flash->success('The category has been moved Up.');
        } else {
            $this->Flash->error('The category could not be moved up. Please, try_
↪again.');
        }
        return $this->redirect($this->referer(['action' => 'index']));
    }

    public function moveDown($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveDown($category)) {
            $this->Flash->success('The category has been moved down.');
        } else {
            $this->Flash->error('The category could not be moved down. Please, try_
↪again.');
        }
        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

Em `src/Template/Categories/index.ctp` substituir o conteúdo existente com:

```
<div class="actions large-2 medium-3 columns">
    <h3><?= __('Actions') ?></h3>
    <ul class="side-nav">
        <li><?= $this->Html->link(__('New Category'), ['action' => 'add']) ?></li>
    </ul>
</div>
<div class="categories index large-10 medium-9 columns">
    <table cellpadding="0" cellspacing="0">
        <thead>
```

```

        <tr>
            <th>Id</th>
            <th>Parent Id</th>
            <th>Lft</th>
            <th>Rght</th>
            <th>Name</th>
            <th>Description</th>
            <th>Created</th>
            <th class="actions"><? __('Actions') ?></th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($categories as $category): ?>
            <tr>
                <td><? $category->id ?></td>
                <td><? $category->parent_id ?></td>
                <td><? $category->lft ?></td>
                <td><? $category->rght ?></td>
                <td><? h($category->name) ?></td>
                <td><? h($category->description) ?></td>
                <td><? h($category->created) ?></td>
                <td class="actions">
                    <? $this->Html->link(__('View'), ['action' => 'view', $category->
↪id]) ?>
                    <? $this->Html->link(__('Edit'), ['action' => 'edit', $category->
↪id]) ?>
                    <? $this->Form->postLink(__('Delete'), ['action' => 'delete',
↪$category->id], ['confirm' => __('Are you sure you want to delete # {0}?',
↪$category->id)]) ?>
                    <? $this->Form->postLink(__('Move down'), ['action' => 'moveDown',
↪$category->id], ['confirm' => __('Are you sure you want to move down # {0}?',
↪$category->id)]) ?>
                    <? $this->Form->postLink(__('Move up'), ['action' => 'moveUp',
↪$category->id], ['confirm' => __('Are you sure you want to move up # {0}?',
↪$category->id)]) ?>
                </td>
            </tr>
        <?php endforeach; ?>
    </tbody>
</table>
</div>

```

Modificando o ArticlesController

Em nossa ArticlesController, vamos obter a lista de todas as categorias. Isto irá permitir-nos para escolher uma categoria para um artigo ao criar ou editar ele:

```

// src/Controller/ArticlesController.php
namespace App\Controller;

use Cake\Network\Exception\NotFoundException;

class ArticlesController extends AppController
{
    // ...

```

```

public function add()
{
    $article = $this->Articles->newEntity();
    if ($this->request->is('post')) {
        $article = $this->Articles->patchEntity($article, $this->request->
        ↪getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Your article has been saved.'));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Unable to add your article.'));
    }
    $this->set('article', $article);

    // Just added the categories list to be able to choose
    // one category for an article
    $categories = $this->Articles->Categories->find('treeList');
    $this->set(compact('categories'));
}
}

```

Modificando os artigos Templates

O artigo adicionado deveria se parecer como isto:

```

<!-- File: src/Template/Articles/add.ctp -->

<h1>Add Article</h1>
<?php
echo $this->Form->create($article);
// just added the categories input
echo $this->Form->input('category_id');
echo $this->Form->input('title');
echo $this->Form->input('body', ['rows' => '3']);
echo $this->Form->button(__('Save Article'));
echo $this->Form->end();

```

Quando você vai para o endereço `/yoursite/categories/add` você deve ver uma lista de categorias para escolher.

Tutorial - Criando um Blog - Autenticação e Autorização

Continuando com o exemplo de *Tutorial - Criando um Blog - Parte 1*, imagine que queríamos garantir o acesso a certas URLs, com base no usuário logado. Temos também uma outra exigência: permitir que o nosso blog para tenha vários autores que podem criar, editar e excluir seus próprios artigos, e bloquear para que outros autores não façam alterações nos artigos que não lhes pertencem.

Criando todo o código relacionado ao Usuário

Primeiro, vamos criar uma nova tabela no banco de dados do blog para armazenar dados de nossos usuários:

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Respeitando as convenções do CakePHP para nomear tabelas, mas também aproveitando de outras convenção: Usando as colunas username e password da tabela de usuários, CakePHP será capaz de configurar automaticamente a maioria das coisas para nós, na implementação do login do usuário.

O próximo passo é criar a nossa classe UsersTable, responsável por encontrar, salvar e validar os dados do usuário:

```
// src/Model/Table/UsersTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class UsersTable extends Table
{
    public function validationDefault(Validator $validator)
    {
        return $validator
            ->notEmpty('username', 'Usuário é necessário')
            ->notEmpty('password', 'Senha é necessária')
            ->notEmpty('role', 'Função é necessária')
            ->add('role', 'inList', [
                'rule' => ['inList', ['admin', 'author']],
                'message' => 'Por favor informe uma função válida'
            ]);
    }
}
```

Vamos também criar o nosso UsersController. O conteúdo a seguir corresponde a partes de uma classe UsersController básica gerado através do utilitário de geração de código bake fornecido com CakePHP:

```
// src/Controller/UsersController.php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\Event;

class UsersController extends AppController
{
    public function beforeFilter(Event $event)
    {
        parent::beforeFilter($event);
        $this->Auth->allow('add');
    }

    public function index()
```

```

    {
        $this->set('users', $this->Users->find('all'));
    }

    public function view($id)
    {
        $user = $this->Users->get($id);
        $this->set(compact('user'));
    }

    public function add()
    {
        $user = $this->Users->newEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__('O usuário foi salvo.'));
                return $this->redirect(['action' => 'add']);
            }
            $this->Flash->error(__('Não é possível adicionar o usuário.'));
        }
        $this->set('user', $user);
    }
}

```

Da mesma maneira que criamos as views para os nossos artigos usando a ferramenta de geração de código, podemos implementar as views do usuário. Para o propósito deste tutorial, vamos mostrar apenas o add.ctp:

```

<!-- src/Template/Users/add.ctp -->

<div class="users form">
    <?=$this->Form->create($user) ?>
    <fieldset>
        <legend><?=__('Add User') ?></legend>
        <?=$this->Form->input('username') ?>
        <?=$this->Form->input('password') ?>
        <?=$this->Form->input('role', [
            'options' => ['admin' => 'Admin', 'author' => 'Author']
        ]) ?>
    </fieldset>
    <?=$this->Form->button(__('Submit')); ?>
    <?=$this->Form->end() ?>
</div>

```

Autenticação (Login e Logout)

Agora estamos prontos para adicionar a nossa camada de autenticação. Em CakePHP isso é tratado pelo `Cake\Controller\Component\AuthComponent`, uma classe responsável por exigir o login para determinadas ações, a manipulação de login e logout de usuário, e também permite as ações para que estão autorizados.

Para adicionar este componente em sua aplicação abra o arquivo `src/Controller/AppController.php` e adicione as seguintes linha:

```
// src/Controller/AppController.php
```

```
namespace App\Controller;

use Cake\Controller\Controller;
use Cake\Event\Event;

class AppController extends Controller
{
    //...

    public function initialize()
    {
        $this->loadComponent('Flash');
        $this->loadComponent('Auth', [
            'loginRedirect' => [
                'controller' => 'Articles',
                'action' => 'index'
            ],
            'logoutRedirect' => [
                'controller' => 'Pages',
                'action' => 'display',
                'home'
            ]
        ]);
    }

    public function beforeFilter(Event $event)
    {
        $this->Auth->allow(['index', 'view', 'display']);
    }
    //...
}
```

Não há muito para ser configurado, como usamos as convenções para a tabela de usuários. Nós apenas configuramos as URLs que serão carregados após o login e logout, estas ações são realizadas no nosso caso para os `/articles/` e `/` respectivamente.

O que fizemos na função `beforeFilter()` foi dizer ao `AuthComponent` para não exigir login em todos `index()` e `view()`, em cada controlador. Queremos que os nossos visitantes sejam capaz de ler e listar as entradas sem registrar-se no site.

Agora, precisamos ser capaz de registrar novos usuários, salvar seu `username` e `password`, e mais importante, o hash da senha para que ele não seja armazenado como texto simples no nosso banco de dados. Vamos dizer ao `AuthComponent` para permitir que usuários deslogados acessem a função `add` e execute as ações de login e logout:

```
// src/Controller/UsersController.php

public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
    // Permitir aos usuários se registrarem e efetuar logout.
    // Você não deve adicionar a ação de "login" a lista de permissões.
    // Isto pode causar problemas com o funcionamento normal do AuthComponent.
    $this->Auth->allow(['add', 'logout']);
}

public function login()
{

```

```

    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            return $this->redirect($this->Auth->redirectUrl());
        }
        $this->Flash->error(__('Usuário ou senha inválido, tente novamente'));
    }
}

public function logout()
{
    return $this->redirect($this->Auth->logout());
}

```

O hashing da senha ainda não está feito, precisamos de uma classe a fim de manipular sua geração. Crie o arquivo **src/Model/Entity/User.php** e adicione a seguinte trecho:

```

// src/Model/Entity/User.php
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
{
    // Gera conjunto de todos os campos exceto o com a chave primária.
    protected $_accessible = [
        '*' => true,
        'id' => false
    ];

    // ...

    protected function _setPassword($password)
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher)->hash($password);
        }
    }

    // ...
}

```

Agora, a senha criptografada usando a classe `DefaultPasswordHasher`. Está faltando apenas o arquivo para exibição da tela de login. Abra o arquivo **src/Template/Users/login.ctp** e adicione as seguintes linhas:

```

<!-- File: src/Template/Users/login.ctp -->

<div class="users form">
<?= $this->Flash->render('auth') ?>
<?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Por favor informe seu usuário e senha') ?></legend>
        <?= $this->Form->input('username') ?>
        <?= $this->Form->input('password') ?>
    </fieldset>

```

```
<?= $this->Form->button(__('Login')); ?>
<?= $this->Form->end() ?>
</div>
```

Agora você pode registrar um novo usuário, acessando a URL `/users/add` e faça login com o usuário recém-criado, indo para a URL `/users/login`. Além disso, tente acessar qualquer outro URL que não tenha sido explicitamente permitido, como `/articles/add`, você vai ver que o aplicativo redireciona automaticamente para a página de login.

E é isso! Parece simples demais para ser verdade. Vamos voltar um pouco para explicar o que aconteceu. A função `beforeFilter()` está falando para o `AuthComponent` não solicitar um login para a ação `add()` em adição as ações `index()` e `view()` que foram prontamente autorizadas na função `beforeFilter()` do `AppController`.

A ação `login()` chama a função `$this->Auth->identify()` da `AuthComponent`, que funciona sem qualquer outra configuração porque estamos seguindo convenções, como mencionado anteriormente. Ou seja, ter uma tabela de usuários com um `username` e uma coluna de `password`, e usamos um form para postar os dados do usuário para o controller. Esta função retorna se o login foi bem sucedido ou não, e caso ela retorne sucesso, então nós redirecionamos o usuário para a URL que configuramos quando adicionamos o `AuthComponent` em nossa aplicação.

O logout funciona quando acessamos a URL `/users/logout` que irá redirecionar o usuário para a url configurada em `logoutUrl`. Essa url é acionada quando a função `AuthComponent::logout()`.

Autorização (quem tem permissão para acessar o que)

Como afirmado anteriormente, nós estamos convertendo esse blog em uma ferramenta multi usuário de autoria, e para fazer isso, precisamos modificar a tabela de artigos um pouco para adicionar a referência à tabela de Usuários:

```
ALTER TABLE articles ADD COLUMN user_id INT(11);
```

Além disso, uma pequena mudança no `ArticlesController` é necessário para armazenar o usuário conectado no momento como uma referência para o artigo criado:

```
// src/Controller/ArticlesController.php

public function add()
{
    $article = $this->Articles->newEntity();
    if ($this->request->is('post')) {
        $article = $this->Articles->patchEntity($article, $this->request->getData());
        // Adicione esta linha
        $article->user_id = $this->Auth->user('id');
        // Você também pode fazer o seguinte
        //$newData = ['user_id' => $this->Auth->user('id')];
        //$article = $this->Articles->patchEntity($article, $newData);
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Seu artigo foi salvo.));
            return $this->redirect(['action' => 'index']);
        }
        $this->Flash->error(__('Não foi possível adicionar seu artigo.));
    }
    $this->set('article', $article);
}
```

A função `user()` fornecida pelo componente retorna qualquer coluna do usuário logado no momento. Nós usamos esse método para adicionar a informação dentro de request data para que ela seja salva.

Vamos garantir que nossa app evite que alguns autores editem ou apaguem posts de outros. Uma regra básica para nossa aplicação é que usuários admin possam acessar qualquer url, enquanto usuários normais (o papel author) podem somente acessar as actions permitidas. Abra novamente a classe `AppController` e adicione um pouco mais de opções para as configurações do Auth:

```
// src/Controller/AppController.php

public function initialize()
{
    $this->loadComponent('Flash');
    $this->loadComponent('Auth', [
        'authorize' => ['Controller'], // Adicione esta linha
        'loginRedirect' => [
            'controller' => 'Articles',
            'action' => 'index'
        ],
        'logoutRedirect' => [
            'controller' => 'Pages',
            'action' => 'display',
            'home'
        ]
    ]);
}

public function isAuthorized($user)
{
    // Admin pode acessar todas as actions
    if (isset($user['role']) && $user['role'] === 'admin') {
        return true;
    }

    // Bloqueia acesso por padrão
    return false;
}
```

Acabamos de criar um mecanismo de autorização muito simples. Nesse caso os usuários com papel admin poderão acessar qualquer url no site quando estiverem logados, mas o restante dos usuários (author) não podem acessar qualquer coisa diferente dos usuários não logados.

Isso não é exatamente o que nós queremos, por isso precisamos corrigir nosso método `isAuthorized()` para fornecer mais regras. Mas ao invés de fazer isso no `AppController`, vamos delegar a cada controller para suprir essas regras extras. As regras que adicionaremos para o `add` de `ArticlesController` deve permitir ao autores criarem os posts mas evitar a edição de posts que não sejam deles. Abra o arquivo `src/Controller/ArticlesController.php` e adicione o seguinte conteúdo:

```
// src/Controller/ArticlesController.php

public function isAuthorized($user)
{
    // Todos os usuários registrados podem adicionar artigos
    if ($this->request->getParam('action') === 'add') {
        return true;
    }

    // Apenas o proprietário do artigo pode editar e excluir
    if (in_array($this->request->getParam('action'), ['edit', 'delete'])) {
        $articleId = (int)$this->request->getParam('pass.0');
        if ($this->Articles->isOwnedBy($articleId, $user['id'])) {

```

```
        return true;
    }
}

return parent::isAuthorized($user);
}
```

Estamos sobrescrevendo a chamada `isAuthorized()` do `AppController` e internamente verificando na classe pai se o usuário está autorizado. Caso não esteja, então apenas permitem acessar a action `add`, e condicionalmente action `edit` e `delete`. Uma última coisa não foi implementada. Para dizer ou não se o usuário está autorizado a editar o artigo, nós estamos chamando uma função `isOwnedBy()` na tabela artigos. Vamos, então, implementar essa função:

```
// src/Model/Table/ArticlesTable.php

public function isOwnedBy($articleId, $userId)
{
    return $this->exists(['id' => $articleId, 'user_id' => $userId]);
}
```

Isso conclui então nossa autorização simples e nosso tutorial de autorização. Para garantir o `UsersController` você pode seguir as mesmas técnicas que usamos para `ArticlesController`, você também pode ser mais criativo e codificar algumas coisas mais gerais no `AppController` para suas próprias regras baseadas em papéis.

Se precisar de mais controle, nós sugerimos que leia o guia completo do Auth [Authentication](#) seção onde você encontrará mais sobre a configuração do componente, criação de classes de Autorização customizadas, e muito mais.

Sugerimos as seguintes leituras

1. *Geração de código com `bake`* Generating basic CRUD code
2. *Authentication*: User registration and login

Contribuindo

Existem várias maneiras de contribuir com o CakePHP. As seções abaixo irão abordar essas formas de contribuição:

Documentação

Contribuir com a documentação é simples. Os arquivos estão hospedados em <https://github.com/cakephp/docs>. Sinta-se a vontade para copiar o repositório, adicionar suas alterações/melhorias/traduições e enviar um *pull request*. Você também pode editar a documentação online pelo Github, mesmo sem ter que fazer *download* dos arquivos – O botão “IMPROVE THIS DOC” presente na lateral direita em qualquer página vai direcioná-lo para o editor online do Github.

A documentação do CakePHP é *continuamente integrada*³³, sendo assim, você pode checar o status de *várias builds*³⁴ no servidor Jenkins a qualquer momento.

Traduções

Envie um email para o time de documentação (docs at cakephp dot org) ou apareça no IRC (#cakephp na freenode) para discutir sobre qualquer área de tradução que você queira participar.

Nova tradução linguística

Nós queremos oferecer traduções completas tanto quanto possível, porém, podem haver momentos que um arquivo de tradução não está atualizado. Você deve sempre considerar a versão em inglês como a prevalecente.

Se o seu idioma não está dentre os listados, por favor nos contate pelo Github e nós vamos considerar incluí-lo. As seções a seguir são as primeiras que você deve considerar traduzir, pois seus arquivos não mudam frequentemente:

- index.rst
- intro.rst
- quickstart.rst
- installation.rst
- /intro (todo o diretório)

³³ http://en.wikipedia.org/wiki/Continuous_integration

³⁴ <https://ci.cakephp.org>

- /tutorials-and-examples (todo o diretório)

Lembrete para administradores de documentação

A estrutura de todos os diretórios de idioma devem espelhar a estrutura da matriz em inglês. Se a estrutura da documentação inglesa sofrer mudanças, as mesmas devem ser aplicadas em outros idiomas.

Por exemplo, se um novo arquivo é criado em **en/file.rst**, nós devemos:

- Adicionar o arquivo a outros idiomas: **pt/file.rst**, **fr/file.rst**, etc
- Deletar o conteúdo, mas manter as informações `title`, `meta`` e eventuais elementos ```toc-tree`. A nota a seguir será adicionada até que alguém traduza o arquivo:

```
File Title
#####

.. note::
    Atualmente, a documentação desta página não é suportada em português.

    Por favor, sintá-se a vontade para nos enviar um *pull request* para o
    `Github <https://github.com/cakephp/docs>`_ ou use o botão
    **IMPROVE THIS DOC** para propor suas mudanças diretamente.

    Você pode consultar a versão em inglês deste tópico através do seletor de
    idiomas localizado ao lado direito do campo de buscas da documentação.

// Se os elementos toc-tree existirem na versão inglesa
.. toctree::
    :maxdepth: 1

    toc-file-x
    toc-file-y
    toc-file-z

.. meta::
    :title lang=pt: Título do arquivo
    :keywords lang=pt: título, descrição,...
```

Dicas para tradutores

- Navegue pelo idioma para o qual deseja traduzir a fim de certificar-se do que já foi traduzido.
- Sintá-se a vontade para mergulhar na tradução caso o seu idioma já exista no manual.
- Use [Linguagem Informal](https://pt.wikipedia.org/wiki/Linguagem_informal)³⁵.
- Traduza o conteúdo e o título ao mesmo tempo.
- Antes de submeter uma correção, compare à versão original. (se você corrigir algo, mas não indicar uma referência, sua submissão não será aceita).
- Se você precisar escrever um termo em inglês, envolva-o na *tag* ``. E.g. “asdf asdf *Controller* asdf” ou “asdf asdf *Kontroller (Controller)* asfd”, como for apropriado.
- Não submeta traduções incompletas.
- Não edite uma seção com alterações pendentes.

³⁵ https://pt.wikipedia.org/wiki/Linguagem_coloquial

- Não use Entidades HTML³⁶ para caracteres acentuados, o manual usa UTF-8.
- Não faça alterações significativas na marcação (HTML) ou adicione novo conteúdo.
- Se estiver faltando alguma informação no conteúdo original, submeta uma correção para tal antes de incluí-la em seu idioma.

Guia de formatação para documentação

A nova documentação do CakePHP é escrita com ReST³⁷. ReST (*Re Structured Text*) é uma sintaxe de marcação de texto simples, semelhante a *markdown* ou *textfile*. Para manter a consistência, recomenda-se que ao adicionar conteúdo à documentação do CakePHP, você siga as diretrizes aqui exemplificadas.

Comprimento da linha

Linhas de texto devem ser limitadas a 80 colunas. As únicas exceções devem ser URLs longas e trechos de código.

Títulos e Seções

Cabeçalhos de seção são criados ao sublinhar o título com caracteres de pontuação seguindo o comprimento do texto.

- # é usado para títulos de páginas.
- = é usado para seções de página.
- – é usado para sub-seções de página.
- ~ é usado para sub-sub-seções de página.
- ^ é usado para sub-sub-sub-seções de página.

Os títulos não devem ser aninhados por mais de 5 níveis de profundidade. Os títulos devem ser precedidos e seguidos por uma linha em branco.

Parágrafos

Os parágrafos são simplesmente blocos de texto, com todas as linhas no mesmo nível de recuo. Os parágrafos devem ser separados por mais do que uma linha vazia.

Marcação em linha

- Um asterisco: *texto* para dar ênfase (itálico) Vamos usá-lo para realce/ênfase.
 - `*texto*`.
- Dois asteriscos: **texto** para ênfase forte (negrito) Vamos usá-lo para diretórios, títulos de listas, nomes de tabelas (excluindo a palavra “*tabela*”).
 - `**/config/Migrations**, **articles**, etc.`
- Dois *backquotes*: `texto` para exemplos de código Vamos usá-lo para opções, nomes de colunas de tabelas, nomes de objetos (excluindo a palavra “*objeto*”) e nomes de métodos/funções – incluir “()”.

³⁶ http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references

³⁷ <http://en.wikipedia.org/wiki/ReStructuredText>

```
- ``cascadeCallbacks``, ``true``, ``id``, ``PagesController``, ``config()``,  
  etc.
```

Se asteriscos ou *backquotes* aparecerem em texto corrido e ficarem confusos com delimitadores de marcação em linha, eles devem ser escapados com um *backslash*.

Marcação em linha tem algumas restrições:

- **Não deve** estar aninhado.
- O conteúdo não deve começar ou terminar com espaço: `* texto*` está errado.
- O conteúdo deve estar separado de texto adjacente por caracteres *non-word*. Use um espaço escapado com uma contrabarra ao seu redor: `umalonga\ *negrito*\ palavra`.

Listas

A marcação de listas é muito parecida com o *markdown*. Listas desordenadas começam com um asterisco e um espaço. Listas enumeradas podem ser criadas tanto com números, ou # para auto numeração:

```
* Esse é um item  
* Esse também, mas esse tem  
  duas linhas.  
  
1. Primeira linha  
2. Segunda linha  
  
#. Numeração automática  
#. Vai lhe economizar algum tempo...
```

Listas com recuos também podem ser criadas ao recuar seções e separá-las com uma linha em branco:

```
* Primeira linha  
* Segunda linha  
  
    * Mais fundo  
    * WOW!  
  
* De volta ao primeiro nível...
```

Listas de definição podem ser criadas assim:

```
Termo  
  Definição  
CakePHP  
  Um framework MVC para PHP
```

Termos não podem ultrapassar uma linha, porém definições podem e devem estar recuadas consistentemente.

Links

Existem diversos tipos de *links*, cada um com usos particulares.

Links externos

Links para documentos externos podem ser feitos desta forma:

```
`Link externo para php.net <http://php.net>`_
```

O link resultante ficaria assim: [Link externo para php.net](http://php.net)³⁸

Links para outras páginas

:doc:

Outras páginas na documentação podem ser referenciadas ao usar a função `:doc:`. Você pode referenciar páginas usando caminho absoluto ou relativo. Você deve omitir a extensão `.rst`. Por exemplo, se a referência `:doc:`form`` estivesse no documento `core-helpers/html`, então o *link* referenciará `core-helpers/form`. Caso a referência fosse `:doc:`/core-helpers``, iria sempre referenciar `/core-helpers` independente de onde a função fosse usada.

Links de referências cruzados

:ref:

Você pode referenciar qualquer título de um documento usando a função `:ref:`. O título por sua vez, não pode ser repetido por toda a documentação. Ao criar títulos para métodos de classes, é melhor usar `class-method` como formato.

A posição mais comum é a cima de um título. Exemplo:

```
.. _label-name:

Título da seção
-----

Mais conteúdo aqui
```

Em qualquer lugar você pode referenciar a seção a cima usando `:ref:`label-name``. O texto do link deverá ser o título que o *link* precedeu. Você pode indicar qualquer formato usando `:ref:`Seu texto <label-name>``.

Prevenindo alertas do Sphinx

O Sphinx vai disparar alertas se um arquivo não for referenciado em um *toc-tree*. É uma forma de garantir que todos os arquivos possuem um *link* referenciado a eles, mas as vezes, você não precisa inserir um *link* para um arquivo, e.g. para seus arquivos *epub-contents* and *pdf-contents*. Nesses casos, você pode adicionar `:orphan:` no topo do arquivo, para suprimir alertas.

Descrevendo classes e seus conteúdos

A documentação do CakePHP usa o [phpdomain](http://python.org/pypi/sphinxcontrib-phpdomain)³⁹ para fornecer directivas customizadas a fim de descrever objetos e construtores no PHP. Usar essas directivas e funções é um requisito para gerar a indexação adequada e recursos de referência cruzada.

³⁸ <http://php.net>

³⁹ <http://pypi.python.org/pypi/sphinxcontrib-phpdomain>

Descrevendo classes e construtores

Cada directiva popula o índice, e/ou o índice do *namespace*.

.. **php:global::** name

Esta directiva declara uma nova variável global PHP.

.. **php:function::** name(signature)

Esta directiva define uma nova função global fora de uma classe.

.. **php:const::** name

Esta directiva declara uma nova constante PHP, você também pode usá-lo aninhada dentro de uma directiva de classe para criar constantes de classe.

.. **php:exception::** name

Esta directiva declara uma nova exceção no *namespace* atual. A assinatura pode incluir argumentos do construtor.

.. **php:class::** name

Esta directiva descreve uma classe. Métodos, atributos, e as constantes pertencentes à classe devem estar dentro do corpo desta directiva:

```
.. php:class:: MyClass

    Descrição da classe

    .. php:method:: method($argument)

        Descrição do método

Atributos, métodos e constantes não precisam estar aninhados. Eles podem
apenas seguir a declaração da classe::

.. php:class:: MyClass

    Texto sobre a classe

    .. php:method:: methodName()

        Texto sobre o método
```

Ver também:

[*php:method*](#), [*php:attr*](#), [*php:const*](#)

.. **php:method::** name(signature)

Descreve um método de classe, seus argumentos, valor de retorno e exceções:

```
.. php:method:: instanceMethod($one, $two)

    :param string $one: O primeiro parâmetro.
    :param string $two: O segundo parâmetro.
    :returns: Um vetor de coisas.
    :throws: InvalidArgumentException

    Este é um método de instância
```

.. **php:staticmethod::** ClassName::methodName(signature)

Descreve um método estático, seus argumentos, valor de retorno e exceções. Ver [*php:method*](#) para opções.


```
.. php:attr:: name
```

Descreve uma propriedade/atributo numa classe.

Prevenindo alertas do Sphinx

O Sphinx vai disparar alertas se uma função estiver referenciada em múltiplos arquivos. É um meio de garantir que você não adicionou uma função duas vezes, porém, algumas vezes você quer escrever a função em dois ou mais arquivos, e.g. *debug object* está referenciado em */development/debugging* e em */core-libraries/global-constants-and-functions*. Nesse caso, você pode adicionar `:noindex:` abaixo do *debug* da função para suprimir alertas. Mantenha apenas uma referência **sem** `:no-index:` para preservar a função referenciada:

```
.. php:function:: debug(mixed $var, boolean $showHtml = null, $showFrom = true)
    :noindex:
```

Referenciamento cruzado

As funções a seguir se referem a objetos PHP e os *links* são gerados se uma directiva correspondente for encontrada:

:php:func:
Referencia uma função PHP.

:php:global:
Referencia uma variável global cujo nome possui o prefixo \$.

:php:const:
Referencia tanto uma constante global como uma constante de classe. Constantes de classe devem ser precedidas pela classe mãe:

```
DateTime possui uma constante :php:const:`DateTime::ATOM`.
```

:php:class:
Referencia uma classe por nome:

```
:php:class:`ClassName`
```

:php:meth:
Referencia um método de uma classe. Essa função suporta ambos os métodos:

```
:php:meth:`DateTime::setDate`
:php:meth:`Classname::staticMethod`
```

:php:attr:
Referencia a propriedade de um objeto:

```
:php:attr:`ClassName::$propertyName`
```

:php:exc:
Referencia uma exceção.

Código-fonte

Blocos de código literais são criados ao finalizar um parágrafo com `::`. O bloco de código literal deve estar recuado, e como todos os parágrafos, estar separado por linhas vazias:

Isto é um parágrafo::

```
while ($i--) {  
    doStuff()  
}
```

Isto é texto novamente.

Texto literal não é modificado ou formatado, com exceção do primeiro nível de recuo que é removido.

Notas e alertas

Muitas vezes há momentos em que você deseja informar o leitor sobre uma dica importante, nota especial ou um perigo potencial. Admoestações no Sphinx são utilizados apenas para isto. Existem cinco tipos de advertências.

- `.. tip::` Dicas são usadas para documentar ou re-iterar informações importantes ou interessantes. O conteúdo da directiva deve ser escrito em sentenças completas e incluir a pontuação adequada.
- `.. note::` Notas são usadas para documentar uma peça importante de informação. O conteúdo da directiva deve ser escrita em sentenças completas e incluir a pontuação adequada.
- `.. warning::` Alertas são usados para documentar obstáculos em potencial, ou informação referente a segurança. O conteúdo da directiva deve ser escrito em sentenças completas e incluir a pontuação adequada.
- `.. versionadded:: X.Y.Z` Admoestações de versão são usados como notas de recursos adicionados em uma versão específica, `X.Y.Z` sendo a versão na qual o dito recurso foi adicionado.
- `.. deprecated:: X.Y.Z` O oposto das admoestações de versão, admoestações de obsolescência são usados para notificar sobre um recurso obsoleto, are used to notify of a deprecated feature, `X.Y.Z` sendo a versão na qual o dito recurso foi abandonado.

Todas as admoestações são feitas da mesma forma:

```
.. note::
```

```
Recuadas e precedido e seguido por uma linha em branco. Assim como um  
parágrafo.
```

Esse texto não é parte da nota.

Exemplos

Dica: Essa é uma dica que você não sabia.

Nota: Você deve prestar atenção aqui.

Aviso: Pode ser perigoso.

Novo na versão 2.6.3: Esse recurso incrível foi adicionado na versão 2.6.3

Obsoleto desde a versão 2.6.3: Esse recurso antigo foi descontinuado na versão 2.6.3

Tickets

Receber *feedback* e ajuda da comunidade em forma de *tickets* é uma parte extremamente importante do processo de desenvolvimento do CakePHP. Todos os tickets estão hospedados no [GitHub](#)⁴⁰.

Reportando bugs

Relatórios de *bugs* bem escritos são muito úteis. Existem algumas medidas que ajudam a criar relatórios de erro melhores:

- **Faça:** [Busque](#)⁴¹ por *tickets* similares para garantir que ninguém reportou algo similar, ou que o erro não tenha sido corrigido.
- **Faça:** Inclua informações detalhadas de **como reproduzir o erro**. Pode ser na forma de um roteiro de testes ou um trecho de código que demonstre o problema. Sem que haja uma forma de reproduzir o problema é pouco provável que seja corrigido.
- **Faça:** Dê o máximo de detalhes sobre o seu ambiente: (SO, versão do PHP, versão do CakePHP).
- **Não faça:** Não use o sistema de *tickets* para sanar dúvidas. O canal de IRC [#cakephp](#) na [Freenode](#)⁴² possui muitos desenvolvedores disponíveis para ajudar a responder suas dúvidas. Também dê uma olhada no [Stack Overflow](#)⁴³.

Reportando problemas de segurança

Se você encontrar um problema de segurança no CakePHP, use o procedimento a seguir ao invés do sistema de relatórios de *bugs* padrão. Envie um email para **security [at] cakephp.org**. Emails enviados para esse endereço são encaminhados para os *core developers* do CakePHP numa lista privada.

Para cada relatório, tentaremos inicialmente confirmar a vulnerabilidade. Uma vez confirmada, o time do CakePHP tomará as ações seguintes:

- Confirmar ao relatante que recebemos o relatório e que estamos trabalhando em um *fix*. Solicitamos ao relatante que o problema seja mantido confidencialmente até que o anunciemos.
- Preparar uma correção/*patch*.
- Preparar um *post* descrevendo o problema, e possíveis vulnerabilidades.
- Lançar novas versões para todas as versões afetadas.
- Anunciar o problema no anúncio de lançamento.

Código

Patches e *pull requests* são formas de contribuir com código para o CakePHP. *Pull requests* podem ser criados no Github e tem preferência sobre arquivos de *patch* nos comentários dos *tickets*.

⁴⁰ <https://github.com/cakephp/cakephp/issues>

⁴¹ <https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues>

⁴² <https://webchat.freenode.net>

⁴³ <https://stackoverflow.com/questions/tagged/cakephp>

Configuração inicial

Antes de trabalhar em *patches* para o CakePHP, é uma boa ideia configurar seu ambiente. Você vai precisar do seguinte *software*:

- Git
- PHP 5.6.0 ou maior
- PHPUnit 3.7.0 ou maior

Defina suas informações de usuário com seu nome e endereço de email:

```
git config --global user.name 'Bob Barker'
git config --global user.email 'bob.barker@example.com'
```

Nota: Se você é novo no Git, recomendamos que leia o gratuito e excelente manual [ProGit](#)⁴⁴.

Clone o código-fonte do CakePHP do Github:

- Se você não tem uma conta no [GitHub](#)⁴⁵, crie uma.
- Dê *Fork* no repositório do [CakePHP](#)⁴⁶ clicando no botão **Fork**&.

Depois que seu *fork* for feito, clone seu *fork* para sua máquina:

```
git clone git@github.com:SEUNOME/cakephp.git
```

Adicione o repositório original do CakePHP como seu repositório remoto. Você irá usá-lo posteriormente para solicitar atualizações das alterações no repositório do CakePHP. Assim sua versão local estará sempre atualizada:

```
cd cakephp
git remote add upstream git://github.com/cakephp/cakephp.git
```

Agora que você tem o CakePHP configurado você pode definir uma *conexão com o banco de dados* `$test`, e *executar todos os testes*.

Trabalhando em um patch

Toda vez que for trabalhar em um *bug*, *feature* ou melhoria, crie um *branch* específico.

O *branch* criado deve ser baseado na versão que deseja atualizar. Por exemplo, se você estiver corrigindo um *bug* na versão 3.x, você deve usar o *branch* `master` como base. Se sua alteração for uma correção de *bug* para a versão 2.x, você deve usar o *branch* `2.x`. Isso faz o *merging* das suas alterações uma tarefa muito mais simples futuramente:

```
# corrigindo um bug na versão 3.x
git fetch upstream
git checkout -b ticket-1234 upstream/master

# corrigindo um bug na versão 2.x
git fetch upstream
git checkout -b ticket-1234 upstream/2.x
```

⁴⁴ <http://git-scm.com/book/>

⁴⁵ <http://github.com>

⁴⁶ <http://github.com/cakephp/cakephp>

Dica: Use um nome descritivo para o seu *branch*, referenciar o nome do *ticket* ou da *feature* é uma boa convenção, e.g. ticket-1234, feature-awesome

A cima criamos um *branch* local baseado no *branch* do *upstream* (CakePHP) 2.x. Trabalhe na sua correção/atualização e faça quantos *commits* precisar, mas tenha em mente o seguinte:

- Siga as *Padrões de codificação*.
- Adicione um caso de teste para mostrar que o *bug* está corrigido, ou que a nova *feature* funciona.
- Mantenha alguma lógica em seus *commits* e escreva mensagens limpas e coerentes.

Enviando um pull request

Uma vez que suas alterações estiverem concluídas e prontas para serem integradas ao CakePHP, você deve atualizar seu *branch*:

```
# Correção por rebase a partir do topo do branch master
git checkout master
git fetch upstream
git merge upstream/master
git checkout <branch_name>
git rebase master
```

Isso irá requisitar e mesclar quaisquer alterações que aconteceram no CakePHP desde que você começou suas alterações, e então executar *rebase* ou replicar suas alterações no topo da lista atual. Você pode encontrar um conflito durante o *rebase*. Se o *rebase* abortar precocemente, você pode verificar que arquivos são conflitantes usando o comando `git status`. Resolva cada conflito e então continue o *rebase*:

```
git add <nome-do-arquivo> # faça isso para cada arquivo conflitante.
git rebase --continue
```

Verifique se todos os seus testes continuam a passar e então faça *push* do seu *branch* para o seu *fork*:

```
git push origin <nome-do-branch>
```

Se você usou *rebase* após enviar as atualizações do seu *branch* por *push*, você precisará forçar o *push*:

```
git push --force origin <nome-do-branch>
```

Uma vez que o seu *branch* estiver no Github, você pode enviar um *pull request*.

Escolhendo onde suas alterações serão incorporadas

Ao fazer *pull requests* você deve ter certeza que selecionou o *branch* correto, pois você não pode fazer qualquer edição após o *pull request* ter sido criado.

- Se sua alteração for um **bugfix**, não introduzir uma nova funcionalidade e apenas corrigir um comportamento existente que está presente no *release* atual, escolhe o *branch master* como seu alvo.
- Se sua alteração for uma **feature**, então você deve escolher o *branch* referente ao próximo número de versão. Por exemplo, se o *branch* atual estável for 3.2.10, o *branch* a receber novas funcionalidades será o 3.next.
- Se sua alteração quebra funcionalidades existentes, ou API's, então você deverá escolher o próximo *major release*. Por exemplo, se o *branch* estável atual for 3.2.2, então a versão na qual o comportamento pode ser quebrado será na versão 4.x.

Nota: Lembre-se que todo código que você contribui com o CakePHP será licenciado sob a licença MIT, e a [Cake Software Foundation](http://cakefoundation.org/pages/about)⁴⁷ será a proprietária de qualquer código proveniente de contribuição. Os contribuidores devem seguir as [regras comunitárias do CakePHP](http://community.cakephp.org/guidelines)⁴⁸.

Todas as correções de *bugs* incorporadas a um *branch* de manutenção serão posteriormente mescladas nos lançamentos futuros realizados pelo time do CakePHP.

Padrões de codificação

Desenvolvedores do CakePHP deverão usar o [guia de codificação PSR-2](http://www.php-fig.org/psr/psr-2/)⁴⁹ em adição às regras apresentadas a seguir e definidas como padrão.

É recomendado que outros desenvolvedores que optem pelo CakePHP sigam os mesmos padrões.

Você pode usar o [CakePHP Code Sniffer](https://github.com/cakephp/cakephp-codesniffer)⁵⁰ para verificar se o seu código segue os padrões estabelecidos.

Adicionando novos recursos

Nenhum novo recurso deve ser adicionado sem que tenha seus próprios testes definidos, que por sua vez, devem estar passando antes que o novo recurso seja enviado para o repositório.

Indentação

Quatro espaços serão usados para indentação.

Então, teremos uma estrutura similar a:

```
// nível base
    // nível 1
        // nível 2
    // nível 1
// nível base
```

Ou:

```
$booleanVariable = true;
$stringVariable = 'jacaré';
if ($booleanVariable) {
    echo 'Valor booleano é true';
    if ($stringVariable === 'jacaré') {
        echo 'Nós encontramos um jacaré';
    }
}
```

Em situações onde você estiver usando uma função em mais de uma linha, siga as seguintes orientações:

- O parêntese de abertura de uma função multi-linha deve ser o último conteúdo da linha.
- Apenas um argumento é permitido por linha em uma função multi-linha.

⁴⁷ <http://cakefoundation.org/pages/about>

⁴⁸ <http://community.cakephp.org/guidelines>

⁴⁹ <http://www.php-fig.org/psr/psr-2/>

⁵⁰ <https://github.com/cakephp/cakephp-codesniffer>

- O parêntese de fechamento de uma função multi-linha deve ter uma linha reservada para si.

Um exemplo, ao invés de usar a seguinte formatação:

```
$matches = array_intersect_key($this->_listeners,
    array_flip(preg_grep($matchPattern,
        array_keys($this->_listeners), 0)));
```

Use esta:

```
$matches = array_intersect_key(
    $this->_listeners,
    array_flip(
        preg_grep($matchPattern, array_keys($this->_listeners), 0)
    )
);
```

Comprimento da linha

É recomendado manter as linhas próximas de 100 caracteres no comprimento para melhor leitura do código. As linhas não devem ser mais longas que 120 caracteres.

Resumindo:

- 100 caracteres é o limite recomendado.
- 120 caracteres é o limite máximo.

Estruturas de controle

Estruturas de controle são por exemplo, “if”, “for”, “foreach”, “while”, “switch”, etc. A baixo, um exemplo com “if”:

```
if ((expr_1) || (expr_2)) {
    // ação_1;
} elseif (!(expr_3) && (expr_4)) {
    // ação_2;
} else {
    // ação_padrão;
}
```

- Nas estruturas de controle deve existir 1 (um) espaço antes do primeiro parêntese e 1 (um) espaço entre o último parêntese e a chave de abertura.
- Sempre use chaves nas estruturas de controle, mesmo que não sejam necessárias. Elas melhoram a leitura do código e tendem a causar menos erros lógicos.
- A abertura da chave deve ser posicionada na mesma linha que a estrutura de controle. A chave de fechamento deve ser colocada em uma nova linha e ter o mesmo nível de indentação que a estrutura de controle. O conteúdo de dentro das chaves deve começar em uma nova linha e receber um novo nível de indentação.
- Atribuições em linha não devem ser usadas dentro de estruturas de controle.

```
// errado = sem chaves, declaração mal posicionada
if (expr) declaração;

// errado = sem chaves
```

```
if (expr)
    declaração;

// certo
if (expr) {
    declaração;
}

// errado = atribuição em linha
if ($variable = Class::function()) {
    declaração;
}

// certo
$variable = Class::function();
if ($variable) {
    declaração;
}
```

Operadores ternários

Operadores ternários são admissíveis quando toda a operação ternária se encaixa em uma única linha. Já operações mais longas devem ser divididas em declarações `if` `else`. Operadores ternários nunca devem ser aninhados. Opcionalmente parênteses podem ser usados ao redor da verificação de condição ternária para esclarecer a operação:

```
// Bom, simples e legível
$variable = isset($options['variable']) ? $options['variable'] : true;

// Aninhamento é ruim
$variable = isset($options['variable']) ? isset($options['othervar']) ? true : false :
    false;
```

Arquivos de template

Em arquivos de *template* (arquivos `.ctp`) os desenvolvedores devem usar estruturas de controle por palavra-chave. A legibilidade em arquivos de *template* complexos é muito melhor dessa forma. As estruturas de controle podem tanto estar contidas em grandes blocos de código PHP, ou ainda em *tags* PHP separadas:

```
<?php
if ($isAdmin):
    echo '<p>Você é o usuário administrador.</p>';
endif;
?>
<p>A seguinte estrutura também é aceitável:</p>
<?php if ($isAdmin): ?>
    <p>Você é o usuário administrador.</p>
<?php endif; ?>
```

Comparação

Sempre tente ser o mais rigoroso possível. Se uma comparação deliberadamente não é estrita, pode ser inteligente comentar sobre isso para evitar confusões geradas por falta de informação.

Para testar se uma variável é nula, é recomendado usar uma verificação estrita:

```
if ($value === null) {
    // ...
}
```

O valor a ser verificado deve ser posto do lado direito:

```
// não recomendado
if (null === $this->foo()) {
    // ...
}

// recomendado
if ($this->foo() === null) {
    // ...
}
```

Chamadas de função

Funções devem ser chamadas sem espaço entre o nome da função e o parêntese de abertura. Deve haver um espaço entre cada parâmetro de uma chamada de função:

```
$var = foo($bar, $bar2, $bar3);
```

Como você pode ver a cima, deve haver um espaço em ambos os lados do sinal de igual (=).

Definição de método

Exemplo de uma definição de método:

```
public function someFunction($arg1, $arg2 = '')
{
    if (expr) {
        declaração;
    }
    return $var;
}
```

Parâmetros com um valor padrão, devem ser posicionados por último na definição de uma função. Tente fazer suas funções retornarem algo, pelo menos true ou false, assim pode-se determinar se a chamada de função foi bem-sucedida:

```
public function connection($dns, $persistent = false)
{
    if (is_array($dns)) {
        $dnsInfo = $dns;
    } else {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!$($dnsInfo) || !$($dnsInfo['phpType'])) {
        return $this->addError();
    }
}
```

```
    return true;
}
```

Existem espaços em ambos os lados dos sinais de igual.

Declaração de tipo

Argumentos que esperam objetos, *arrays* ou *callbacks* (válidos) podem ser declarados por tipo. Nós apenas declaramos métodos públicos, porém, o uso da declaração por tipo não é livre de custos:

```
/**
 * Descrição do método.
 *
 * @param \Cake\ORM\Table $table A classe Table a ser usada.
 * @param array $array Algum valor em formato array.
 * @param callable $callback Algum callback.
 * @param bool $boolean Algum valor booleano.
 */
public function foo(Table $table, array $array, callable $callback, $boolean)
{
}
```

Aqui `$table` deve ser uma instância de `\Cake\ORM\Table`, `$array` deve ser um `array` e `$callback` deve ser do tipo `callable` (um *callback* válido).

Perceba que se você quiser permitir `$array` ser também uma instância de `\ArrayObject` você não deve declará-lo, pois `array` aceita apenas o tipo primitivo:

```
/**
 * Descrição do método.
 *
 * @param array|\ArrayObject $array Algum valor em formato array.
 */
public function foo($array)
{
}
```

Funções anônimas (Closures)

Para se definir funções anônimas, segue-se o estilo de codificação [PSR-2](http://www.php-fig.org/psr/psr-2/)⁵¹, onde elas são declaradas com um espaço depois da palavra-chave *function*, e um espaço antes e depois da palavra-chave *use*:

```
$closure = function ($arg1, $arg2) use ($var1, $var2) {
    // código
};
```

Encadeamento de métodos

Encadeamento de métodos deve ter múltiplos métodos distribuídos em linhas separadas e indentados com quatro espaços:

⁵¹ <http://www.php-fig.org/psr/psr-2/>

```
$email->from('foo@exemplo.com')
->to('bar@exemplo.com')
->subject('Uma mensagem legal')
->send();
```

Comentando código

Todos os comentários devem ser escritos em inglês, e devem de forma clara descrever o bloco de código comentado.

Comentários podem incluir as seguintes *tags* do phpDocumentor⁵²:

- `@author`⁵³
- `@copyright`⁵⁴
- `@deprecated`⁵⁵ Usando o formato `@version <vector> <description>`, onde `version` e `description` são obrigatórios.
- `@example`⁵⁶
- `@ignore`⁵⁷
- `@internal`⁵⁸
- `@link`⁵⁹
- `@see`⁶⁰
- `@since`⁶¹
- `@version`⁶²

Tags PhpDoc são muito semelhantes a *tags* JavaDoc no Java. *Tags* são apenas processadas se forem a primeira coisa numa linha de DocBlock, por exemplo:

```
/**
 * Exemplo de tag.
 *
 * @author essa tag é analisada, mas essa versão é ignorada
 * @version 1.0 essa tag também é analisada
 */
```

```
/**
 * Exemplo de tags phpDoc em linha.
 *
 * Essa função cria planos com foo() para conquistar o mundo.
 *
 * @return void
 */
```

⁵² <http://phpdoc.org>

⁵³ <http://phpdoc.org/docs/latest/references/phpdoc/tags/author.html>

⁵⁴ <http://phpdoc.org/docs/latest/references/phpdoc/tags/copyright.html>

⁵⁵ <http://phpdoc.org/docs/latest/references/phpdoc/tags/deprecated.html>

⁵⁶ <http://phpdoc.org/docs/latest/references/phpdoc/tags/example.html>

⁵⁷ <http://phpdoc.org/docs/latest/references/phpdoc/tags/ignore.html>

⁵⁸ <http://phpdoc.org/docs/latest/references/phpdoc/tags/internal.html>

⁵⁹ <http://phpdoc.org/docs/latest/references/phpdoc/tags/link.html>

⁶⁰ <http://phpdoc.org/docs/latest/references/phpdoc/tags/see.html>

⁶¹ <http://phpdoc.org/docs/latest/references/phpdoc/tags/since.html>

⁶² <http://phpdoc.org/docs/latest/references/phpdoc/tags/version.html>

```
function bar()
{

}

/**
 * Função foo.
 *
 * @return void
 */
function foo()
{

}
```

Blocos de comentários, com a exceção do primeiro bloco em um arquivo, devem sempre ser precedidos por uma nova linha.

Tipos de variáveis

Tipos de variáveis para serem usadas em DocBlocks:

Tipo Descrição

mixed Uma variável com múltiplos tipos ou tipo indefinido.

int Variável de tipo *int* (número inteiro).

float Variável de tipo *float* (número decimal).

bool Variável de tipo *bool* (lógico, verdadeiro ou falso).

string Variável de tipo *string* (qualquer valor dentro de ”” ou ‘ ‘).

null Variável de tipo *null*. Normalmente usada em conjunto com outro tipo.

array Variável de tipo *array*.

object Variável de tipo *object*. Um nome específico de classe deve ser usado, se possível.

resource Variável de tipo *resource* (retornado de `mysql_connect()` por exemplo). Lembre-se que quando você especificar o tipo como *mixed*, você deve indicar se o mesmo é desconhecido, ou quais os tipos possíveis.

callable Variável de tipo função.

Você também pode combinar tipos usando o caractere de barra vertical:

```
int|bool
```

Para mais de dois tipos é melhor usar `mixed`.

Ao retornar o próprio objeto, e.g. para encadeamento, use `$this` ao invés:

```
/**
 * Função Foo.
 *
 * @return $this
 */
public function foo()
{
    return $this;
}
```

Incluindo arquivos

include, require, include_once e require_once não tem parênteses:

```
// errado = com parênteses
require_once('ClassFileName.php');
require_once ($class);

// certo = sem parênteses
require_once 'ClassFileName.php';
require_once $class;
```

Ao incluir arquivos com classes ou bibliotecas, use sempre e apenas a função `require_once`⁶³.

Tags do PHP

Use sempre *tags* longas (`<?php ?>`) ao invés de *tags* curtas (`<? ?>`). O *short echo* deve ser usado em arquivos de template (`.ctp`) quando apropriado.

Short Echo

O *short echo* deve ser usado em arquivos de template no lugar de `<?php echo`. Deve também, ser imediatamente seguido por um espaço em branco, a variável ou função a ser chamada pelo `echo`, um espaço em branco e a *tag* de fechamento do PHP:

```
// errado = ponto-e-virgula, sem espaços
<td><?=$name; ?></td>

// certo = sem ponto-e-virgula, com espaços
<td><?= $name ?></td>
```

A partir do PHP 5.4 a *tag short echo* (`<?=>`) não é mais considerada um atalho, estando sempre disponível independentemente da configuração da chave `short_open_tag`.

Convenção de nomenclatura

Funções

Escreva todas as funções no padrão “camelBack”, isto é, com a letra da primeira palavra minúscula e a primeira letra das demais palavras maiúsculas:

```
function longFunctionName()
{
}
```

Classes

Escreva todas as funções no padrão “CamelCase”, isto é, com a primeira letra de cada palavra que compõem o nome da classe maiúscula:

⁶³ http://php.net/require_once

```
class ExampleClass
{
}
```

Variáveis

Nomes de variáveis devem ser tanto curtas como descritivas, o quanto possível. Todas as variáveis devem começar com letra minúscula e seguir o padrão “camelBack” no caso de muitas palavras. Variáveis referenciando objetos devem estar de alguma forma associadas à classe indicada. Exemplo:

```
$user = 'John';
$users = ['John', 'Hans', 'Arne'];

$dispatcher = new Dispatcher();
```

Visibilidade

Use as palavras reservadas do PHP5, *private* e *protected* para indicar métodos e variáveis. Adicionalmente, nomes de métodos e variáveis não-públicos começar com um *underscore* singular (*_*). Exemplo:

```
class A
{
    protected $_iAmAProtectedVariable;

    protected function _iAmAProtectedMethod()
    {
        /* ... */
    }

    private $_iAmAPrivateVariable;

    private function _iAmAPrivateMethod()
    {
        /* ... */
    }
}
```

Endereços para exemplos

Para qualquer URL e endereços de email, use “example.com”, “example.org” e “example.net”, por exemplo:

- Email: someone@example.com
- WWW: <http://www.example.com>
- FTP: <ftp://ftp.example.com>

O nome de domínio “example.com” foi reservado para isso (see [RFC 2606](https://tools.ietf.org/html/rfc2606)⁶⁴), sendo recomendado o seu uso em documentações como exemplos.

⁶⁴ [https://tools.ietf.org/html/rfc2606.html](https://tools.ietf.org/html/rfc2606)

Arquivos

Nomes de arquivos que não contém classes devem ser em caixa baixa e sublinhados, por exemplo:

```
long_file_name.php
```

Moldagem de tipos

Para moldagem usamos:

Tipo Descrição

(bool) Converte para *boolean*.

(int) Converte para *integer*.

(float) Converte para *float*.

(string) Converte para *string*.

(array) Converte para *array*.

(object) Converte para *object*.

Por favor use `(int)$var` ao invés de `intval($var)` e `(float)$var` ao invés de `floatval($var)` quando aplicável.

Constante

Constantes devem ser definidas em caixa alta:

```
define('CONSTANT', 1);
```

Se o nome de uma constante consiste de múltiplas palavras, eles devem ser separados por um *underscore*, por exemplo:

```
define('LONG_NAMED_CONSTANT', 2);
```

Cuidados usando empty()/isset()

Apesar de `empty()` ser uma função simples de ser usada, pode mascarar erros e causar efeitos não intencionais quando `'0'` e `0` são retornados. Quando variáveis ou propriedades já estão definidas, o uso de `empty()` não é recomendado. Ao trabalhar com variáveis, é melhor confiar em coerção de tipo com booleanos ao invés de `empty()`:

```
function manipulate($var)
{
    // Não recomendado, $var já está definida no escopo
    if (empty($var)) {
        // ...
    }

    // Recomendado, use coerção de tipo booleano
    if (!$var) {
        // ...
    }
    if ($var) {
        // ...
    }
}
```

```
}  
}
```

Ao lidar com propriedades definidas, você deve favorecer verificações por `null` sobre verificações por `empty()/isset()`:

```
class Thing  
{  
    private $property; // Definida  
  
    public function readProperty()  
    {  
        // Não recomendado já que a propriedade está definida na classe  
        if (!isset($this->property)) {  
            // ...  
        }  
        // Recomendado  
        if ($this->property === null) {  
            // ...  
        }  
    }  
}
```

Ao trabalhar com *arrays*, é melhor mesclar valores padronizados ao usar verificações por `empty()`. Assim, você se assegura que as chaves necessárias estão definidas:

```
function doWork(array $array)  
{  
    // Mescla valores para remover a necessidade de verificações via empty.  
    $array += [  
        'key' => null,  
    ];  
  
    // Não recomendado, a chave já está definida  
    if (isset($array['key'])) {  
        // ...  
    }  
  
    // Recomendado  
    if ($array['key'] !== null) {  
        // ...  
    }  
}
```

Guia de retrocompatibilidade

Garantir que você possa atualizar suas aplicações facilmente é importante para nós. Por esse motivo, apenas quebramos compatibilidade nos *major releases*. Você deve estar familiarizado com [versionamento semântico](http://semver.org/)⁶⁵, orientação usada em todos os projetos do CakePHP. Resumindo, significa que apenas *major releases* (tais como 2.0, 3.0, 4.0) podem quebrar retrocompatibilidades. *Minor releases* (tais como 2.1, 3.1, 4.1) podem introduzir novos recursos, mas não podem quebrar retrocompatibilidades. *Releases* de correção de *bugs* (tais como 2.1.2, 3.0.1) não incluem novos recursos, são destinados apenas à correção de erros e melhora de desempenho.

⁶⁵ <http://semver.org/>

Nota: O CakePHP começou a seguir o versionamento semântico na versão 2.0.0. Essas regras não se aplicam às versões 1.x.

Para esclarecer que mudanças você pode esperar em cada ciclo de *release*, nós temos mais informações detalhadas para desenvolvedores usando o CakePHP, e para desenvolvedores trabalhando Não CakePHP que ajudam a definir expectativas do que pode ser feito em *minor releases*. *Major releases* podem ter tantas quebras quanto forem necessárias.

Guia de migração

Para cada *major* ou *minor releases*, a equipe do CakePHP vai disponibilizar um guia de migração. Esses guias explicam os novos recursos e qualquer quebra de compatibilidade. Eles podem ser encontrados na seção [Apêndices](#) do manual.

Usando o CakePHP

Se você está construindo sua aplicação com o CakePHP, as orientações a seguir vão demonstrar a estabilidade que você pode esperar.

Interfaces

Com exceção dos *major releases*, interfaces oferecidas pelo CakePHP **não** irão ter alterações em qualquer método. Novos métodos podem ser incluídos, mas nenhum método existente será alterado.

Classes

Classes oferecidas pelo CakePHP podem ser construídas e ter seus métodos públicos e propriedades usados. Não código da aplicação e com exceção de *major releases* a retrocompatibilidade é garantida.

Nota: Algumas classes Não CakePHP são marcadas com a *tag* da documentação da API `@internal`. Essas classes **não** são estáveis e não tem garantias de retrocompatibilidade.

Em *minor releases*, novos métodos podem ser adicionados a classes, e métodos existentes podem passar a receber novos argumentos. Qualquer novo argumento vai ter valores padrões, mas se você sobrescrever métodos com uma assinatura diferente, é possível que você receba erros fatais. Métodos que recebem novos argumentos serão documentados. Não guia de migração correspondente ao *release*.

A tabela a seguir descreve quais casos de uso e que tipo de compatibilidade você pode esperar do CakePHP.

Se você...	Retrocompatibilidade?
Typehint referente à classe	Sim
Criar uma nova instância	Sim
Estender a classe	Sim
Acessar uma propriedade pública	Sim
Chamar um método público	Sim
Estender uma classe e...	
Sobrescrever uma propriedade pública	Sim
Acessar uma propriedade protegida	Não ¹
Sobrescrever uma propriedade protegida	Não ¹
Sobrescrever um método	Não ¹
Chamar um método protegido	Não ¹
Adicionar uma propriedade pública	Não
Adicionar um método público	Não
Adicionar um argumento a um método sobrescrito	Não ¹
Adicionar um valor padrão a um argumento de método existente	Sim

Trabalhando no CakePHP

Se você está ajudando a fazer o CakePHP ainda melhor, por favor, siga as orientações a seguir quando estiver adicionando/alterando funcionalidades:

Em um *minor release* você pode:

Em um <i>minor release</i> você pode...	
Classes	
Remover uma classe	Não
Remover uma interface	Não
Remover um trait	Não
Tornar final	Não
Tornar abstract	Não
Trocar o nome	Sim ²
Properties	
Adicionar uma propriedade pública	Sim
Remove a public property	Não
Adicionar uma propriedade protegida	Sim
Remover uma propriedade protegida	Sim ³
Métodos	
Adicionar um método público	Sim
Remover um método público	Não
Adicionar um método público	Sim
Mover para uma classe parente	Sim
Remover um método protegido	Sim ³
Reduzir visibilidade	Não
Mudar nome do método	Sim ²
Adicionar um novo argumento com valor padrão	Sim
Adicionar um novo argumento a um método existente.	Não
Remover um valor padrão de um argumento existente	Não

¹ Seu código *pode* ser quebrado por *minor releases*. Verifique o guia de migração para mais detalhes.

² Você pode mudar o nome de uma classe/método desde que o nome antigo permaneça disponível. Isso normalmente é evitado, a não ser que a renomeação traga algum benefício significativo.

³ Evite sempre que possível. Qualquer remoção precisa ser documentada no guia de migração.

Instalação

O CakePHP é rápido e fácil de instalar. Os requisitos mínimos são um servidor web e uma cópia do CakePHP, só isso! Apesar deste manual focar principalmente na configuração do Apache (porquê ele é o mais simples de instalar e configurar), o CakePHP vai ser executado em uma série de servidores web como nginx, LightHTTPD, ou Microsoft IIS.

Requisitos

- HTTP Server. Por exemplo: Apache. De preferência com mod_rewrite ativo, mas não é obrigatório.
- PHP 5.6.0 ou superior.
- extensão mbstring
- extensão intl

Nota: Tanto no XAMPP quanto no WAMP, as extensões mcrypt e mbstring são setadas por padrão.

Se você estiver usando o XAMPP, já tem a extensão intl inclusa, mas é preciso descomentar a linha `extension=php_intl.dll` no arquivo `php.ini` e então, reiniciar o servidor através do painel de controle do XAMPP.

Caso você esteja usando o WAMP, a extensão intl está “ativa” por padrão, mas não está funcional. Para fazê-la funcionar, você deve ir à pasta do php (que por padrão é) `C:\wamp\bin\php\php{version}`, copiar todos os arquivos que se pareçam com `icu***.dll` e colá-los no diretório “bin” do apache `C:\wamp\bin\apache\apache{version}\bin`. Reiniciando todos os serviços a extensão já deve ficar ok.

Apesar de um mecanismo de banco de dados não ser exigido, nós imaginamos que a maioria das aplicações irá utilizar um. O CakePHP suporta uma variedade de mecanismos de armazenamento de banco de dados:

- MySQL (5.1.10 ou superior)
- PostgreSQL
- Microsoft SQL Server (2008 ou superior)
- SQLite 3

Nota: Todos os drivers incluídos internamente requerem PDO. Você deve assegurar-se que possui a extensão PDO correta instalada.

Instalando o CakePHP

O CakePHP utiliza [Composer](#)⁶⁶, uma ferramenta de gerenciamento de dependências para PHP 5.3+, como o método suportado oficial para instalação.

Primeiramente, você precisará baixar e instalar o Composer se não o fez anteriormente. Se você tem cURL instalada, é tão fácil quanto executar o seguinte:

```
curl -s https://getcomposer.org/installer | php
```

Ou, você pode baixar `composer.phar` do [Site oficial do Composer](#)⁶⁷.

Para sistemas Windows, você pode baixar o instalador [aqui](#)⁶⁸. Mais instruções para o instalador Windows do Composer podem ser encontradas dentro do LEIA-ME [aqui](#)⁶⁹.

Agora que você baixou e instalou o Composer, você pode receber uma nova aplicação CakePHP executando:

```
php composer.phar create-project --prefer-dist cakephp/app [app_name]
```

Ou se o Composer estiver instalado globalmente:

```
composer create-project --prefer-dist cakephp/app [app_name]
```

Uma vez que o Composer terminar de baixar o esqueleto da aplicação e o núcleo da biblioteca CakePHP, você deve ter uma aplicação funcional instalada via Composer. Esteja certo de manter os arquivos `composer.json` e `composer.lock` com o restante do seu código fonte.

You can now visit the path to where you installed your CakePHP application and see the setup traffic lights.

Mantendo sincronização com as últimas alterações no CakePHP

Se você quer se manter atualizado com as últimas mudanças no CakePHP, você pode adicionar o seguinte ao `composer.json` de sua aplicação:

```
"require": {  
    "cakephp/cakephp": "dev-master"  
}
```

Onde `<branch>` é o nome do branch que você segue. Toda vez que você executar `php composer.phar update` você receberá as últimas atualizações do branch escolhido.

⁶⁶ <http://getcomposer.org>

⁶⁷ <https://getcomposer.org/download/>

⁶⁸ <https://github.com/composer/windows-setup/releases/>

⁶⁹ <https://github.com/composer/windows-setup>

Permissões

O CakePHP utiliza o diretório `tmp` para diversas operações. Descrição de models, views armazenadas em cache e informações de sessão são apenas alguns exemplos. O diretório `logs` é utilizado para escrever arquivos de log pelo mecanismo padrão `FileLog`.

Como tal, certifique-se que os diretórios `logs`, `tmp` e todos os seus sub-diretórios em sua instalação CakePHP são graváveis pelo usuário relacionado ao servidor web. O processo de instalação do Composer faz `tmp` e seus sub-diretórios globalmente graváveis para obter as coisas funcionando rapidamente, mas você pode atualizar as permissões para melhor segurança e mantê-los graváveis apenas para o usuário relacionado ao servidor web.

Um problema comum é que os diretórios e sub-diretórios de `logs` e `tmp` devem ser graváveis tanto pelo servidor quanto pelo usuário da linha de comando. Em um sistema UNIX, se seu usuário relacionado ao servidor web é diferente do seu usuário da linha de comando, você pode executar somente uma vez os seguintes comandos a partir do diretório da sua aplicação para assegurar que as permissões serão configuradas corretamente:

```
HTTPDUSER=`ps aux | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v_\`
↪root | head -1 | cut -d\ -f1`
setfacl -R -m u:${HTTPDUSER}:rwx tmp
setfacl -R -d -m u:${HTTPDUSER}:rwx tmp
setfacl -R -m u:${HTTPDUSER}:rwx logs
setfacl -R -d -m u:${HTTPDUSER}:rwx logs
```

Servidor de Desenvolvimento

Uma instalação de desenvolvimento é o método mais rápido de configurar o CakePHP. Neste exemplo, nós vamos utilizar o console CakePHP para executar o servidor integrado do PHP que vai tornar sua aplicação disponível em `http://host:port`. A partir do diretório da aplicação, execute:

```
bin/cake server
```

Por padrão, sem nenhum argumento fornecido, isso vai disponibilizar a sua aplicação em `http://localhost:8765/`.

Se você tem algo conflitante com `localhost` ou porta 8765, você pode dizer ao console CakePHP para executar o servidor web em um host e/ou porta específica utilizando os seguintes argumentos:

```
bin/cake server -H 192.168.13.37 -p 5673
```

Isto irá disponibilizar sua aplicação em `http://192.168.13.37:5673/`.

É isso aí! Sua aplicação CakePHP está instalada e funcionando sem ter que configurar um servidor web.

Aviso: O servidor de desenvolvimento *nunca* deve ser usado em um ambiente de produção. Destina-se apenas como um servidor de desenvolvimento básico.

Se você preferir usar um servidor web real, você deve ser capaz de mover a instalação do CakePHP (incluindo os arquivos ocultos) para dentro do diretório raiz do seu servidor web. Você deve, então, ser capaz de apontar seu navegador para o diretório que você moveu os arquivos para dentro e ver a aplicação em ação.

Produção

Uma instalação de produção é uma forma mais flexível de configurar o CakePHP. Usar este método permite total domínio para agir como uma aplicação CakePHP singular. Este exemplo o ajudará a instalar o CakePHP em qualquer lugar em seu sistema de arquivos e torná-lo disponível em <http://www.example.com>. Note que esta instalação pode exigir os direitos de alterar o `DocumentRoot` em servidores web Apache.

Depois de instalar a aplicação usando um dos métodos acima no diretório de sua escolha - vamos supor que você escolheu `/cake_install` - sua configuração de produção será parecida com esta no sistema de arquivos:

```
/cake_install/  
  bin/  
  config/  
  logs/  
  plugins/  
  src/  
  tests/  
  tmp/  
  vendor/  
  webroot/ (esse diretório é definido como DocumentRoot)  
  .gitignore  
  .htaccess  
  .travis.yml  
  composer.json  
  index.php  
  phpunit.xml.dist  
  README.md
```

Desenvolvedores utilizando Apache devem definir a diretiva `DocumentRoot` pelo domínio para:

```
DocumentRoot /cake_install/webroot
```

Se o seu servidor web está configurado corretamente, agora você deve encontrar sua aplicação CakePHP acessível em <http://www.example.com>.

Aquecendo

Tudo bem, vamos ver o CakePHP em ação. Dependendo de qual configuração você usou, você deve apontar seu navegador para <http://example.com/> ou <http://localhost:8765/>. Nesse ponto, você será apresentado à página home padrão do CakePHP e uma mensagem que diz a você o estado da sua conexão atual com o banco de dados.

Parabéns! Você está pronto para *create your first CakePHP application*.

Reescrita de URL

Apache

Apesar do CakePHP ser construído para trabalhar com `mod_rewrite` fora da caixa, e normalmente o faz, nos atentamos que alguns usuários lutam para conseguir fazer tudo funcionar bem em seus sistemas.

Aqui estão algumas coisas que você poderia tentar para conseguir tudo rodando corretamente. Primeiramente observe seu `httpd.conf`. (Tenha certeza que você está editando o `httpd.conf` do sistema ao invés de um usuário, ou site específico.)

Esses arquivos podem variar entre diferentes distribuições e versões do Apache. Você também pode pesquisar em <http://wiki.apache.org/httpd/DistrosDefaultLayout> para maiores informações.

1. Tenha certeza que a sobreescrita do .htaccess está permitida e que AllowOverride está definido para All no correto DocumentRoot. Você deve ver algo similar a:

```
# Cada diretório ao qual o Apache tenha acesso pode ser configurado com respeito
# a quais serviços e recursos estão permitidos e/ou desabilitados neste
# diretório (e seus sub-diretórios).
#
# Primeiro, nós configuramos o "default" para ser um conjunto bem restrito de
# recursos.
<Directory />
    Options FollowSymLinks
    AllowOverride All
    # Order deny,allow
    # Deny from all
</Directory>
```

2. Certifique-se que o mod_rewrite está sendo carregado corretamente. Você deve ver algo como:

```
LoadModule rewrite_module libexec/apache2/mod_rewrite.so
```

Em muitos sistemas estará comentado por padrão, então você pode apenas remover os símbolos #.

Depois de fazer as mudanças, reinicie o Apache para certificar-se que as configurações estão ativas.

Verifique se os seus arquivos .htaccess estão realmente nos diretórios corretos. Alguns sistemas operacionais tratam arquivos iniciados com '.' como ocultos e portanto, não os copia.

3. Certifique-se de sua cópia do CakePHP vir da seção de downloads do site ou do nosso repositório Git, e que foi descompactado corretamente, verificando os arquivos .htaccess.

O diretório app do CakePHP (será copiado para o diretório mais alto de sua aplicação através do bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule ^$ webroot/ [L]
    RewriteRule (.*?) webroot/$1 [L]
</IfModule>
```

O diretório webroot do CakePHP (será copiado para a raiz de sua aplicação através do bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Se o seu site CakePHP ainda possuir problemas com mod_rewrite, você pode tentar modificar as configurações para Virtual Hosts. No Ubuntu, edita o arquivo /etc/apache2/sites-available/default (a localização depende da distribuição). Nesse arquivo, certifique-se que AllowOverride None seja modificado para AllowOverride All, então você terá:

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
```

```
Options Indexes FollowSymLinks MultiViews
AllowOverride All
Order Allow,Deny
Allow from all
</Directory>
```

No macOS, outra solução é usar a ferramenta [virtualhostx](http://clickontyler.com/virtualhostx/)⁷⁰ para fazer um Virtual Host apontar para o seu diretório.

Para muitos serviços de hospedagem (GoDaddy, land1), seu servidor web é na verdade oferecido a partir de um diretório de usuário que já utiliza `mod_rewrite`. Se você está instalando o CakePHP em um diretório de usuário (<http://example.com/~username/cakephp/>), ou qualquer outra estrutura URL que já utilize `mod_rewrite`, você precisará adicionar declarações `RewriteBase` para os arquivos `.htaccess` que o CakePHP utiliza. (`.htaccess`, `webroot/.htaccess`).

Isso pode ser adicionado na mesma seção com a diretiva `RewriteEngine`, por exemplo, seu arquivo `webroot/.htaccess` ficaria como:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Os detalhes dessas mudanças vão depender da sua configuração, e podem incluir coisas adicionais que não estão relacionadas ao CakePHP. Por favor, busque pela documentação online do Apache para mais informações.

4. (Opcional) Para melhorar a configuração de produção, você deve prevenir conteúdos inválidos de serem analisados pelo CakePHP. Modifique seu `webroot/.htaccess` para algo como:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app/
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^(webroot/)?(img|css|js)/(.*)$
    RewriteRule ^ index.php [L]
</IfModule>
```

Isto irá simplesmente prevenir conteúdo incorreto de ser enviado para o `index.php` e então exibir sua página de erro 404 do servidor web.

Adicionalmente você pode criar uma página HTML de erro 404 correspondente, ou utilizar a padrão do CakePHP ao adicionar uma diretiva `ErrorDocument`:

```
ErrorDocument 404 /404-not-found
```

nginx

nginx não utiliza arquivos `.htaccess` como o Apache, então é necessário criar as reescritas de URL na configuração de sites disponíveis. Dependendo da sua configuração, você precisará modificar isso, mas pelo menos, você vai precisar do PHP rodando como uma instância FastCGI:

⁷⁰ <http://clickontyler.com/virtualhostx/>


```

server {
    listen    80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com$1 permanent;
}

server {
    listen    80;
    server_name example.com;

    # root directive should be global
    root     /var/www/example.com/public/webroot/;
    index    index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ /\.php$ {
        try_files $uri =404;
        include /etc/nginx/fastcgi_params;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}

```

IIS7 (Windows hosts)

IIS7 não suporta nativamente arquivos .htaccess. Mesmo existindo add-ons que adicionam esse suporte, você também pode importar as regras .htaccess no IIS para utilizar as reescritas nativas do CakePHP. Para isso, siga os seguintes passos:

1. Utilize o [Microsoft's Web Platform Installer](http://www.microsoft.com/web/downloads/platform.aspx)⁷¹ para instalar o [Rewrite Module 2.0](#)⁷² ou baixe-o diretamente (32-bit⁷³ / 64-bit⁷⁴).
2. Crie um novo arquivo chamado web.config em seu diretório raiz do CakePHP.
3. Utilize o Notepad ou qualquer editor seguro XML para copiar o seguinte código em seu novo arquivo web.config:

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <rule name="Exclude direct access to webroot/*"
          stopProcessing="true">
          <match url="^webroot/(.*)$" ignoreCase="false" />
          <action type="None" />

```

⁷¹ <http://www.microsoft.com/web/downloads/platform.aspx>

⁷² <http://www.iis.net/downloads/microsoft/url-rewrite>

⁷³ <http://www.microsoft.com/en-us/download/details.aspx?id=5747>

⁷⁴ <http://www.microsoft.com/en-us/download/details.aspx?id=7435>

```

        </rule>
        <rule name="Rewrite routed access to assets(img, css, files, js,
→favicon)"
            stopProcessing="true">
            <match url="^(img|css|files|js|favicon.ico) (.*)$" />
            <action type="Rewrite" url="webroot/{R:1}{R:2}"
                appendQueryString="false" />
            </rule>
            <rule name="Rewrite requested file/folder to index.php"
                stopProcessing="true">
                <match url="^(.*)$" ignoreCase="false" />
                <action type="Rewrite" url="index.php"
                    appendQueryString="true" />
            </rule>
        </rules>
    </rewrite>
</system.webServer>
</configuration>

```

Uma vez que o arquivo `web.config` é criado com as regras amigáveis de reescrita do IIS, os links, CSS, JavaScript, e roteamento do CakePHP agora devem funcionar corretamente.

Não posso utilizar Reescrita de URL

Se você não quer ou não pode ter `mod_rewrite` (ou algum outro módulo compatível) funcionando no seu servidor, você precisará utilizar as URLs amigáveis nativas do CakePHP. No **`config/app.php`**, descomente a linha que se parece como:

```

'App' => [
    // ...
    // 'baseUrl' => env('SCRIPT_NAME'),
]

```

Também remova esses arquivos `.htaccess`:

```

/.htaccess
webroot/.htaccess

```

Isso fará suas URLs parecerem como `www.example.com/index.php/controllername/actionname/param` ao invés de `www.example.com/controllername/actionname/param`.

Configuração

Embora as convenções eliminem a necessidade de configurar todo o CakePHP, Você ainda precisará configurar algumas coisas, como suas credenciais de banco de dados por exemplo.

Além disso, há opções de configuração opcionais que permitem trocar valores padrão e implementações com as personalizadas para seu aplicativo.

Configurando sua Aplicação

A configuração é geralmente armazenada em arquivos PHP ou INI, e carregada durante a execução do código de inicialização. O CakePHP vem com um arquivo de configuração por padrão. Mas se necessário, você pode adicionar arquivos de configuração adicionais e carregá-los no código de inicialização do aplicativo. `Cake\Core\Configure` é usado para configuração global, e classes como `Cache` providenciam `config()` métodos para tornar a configuração simples e transparente.

Carregando Arquivos de Configurações Adicionais

Se sua aplicação tiver muitas opções de configuração, pode ser útil dividir a configuração em vários arquivos. Depois de criar cada um dos arquivos no seu **config/** diretório, você pode carregá-los em **bootstrap.php**:

```
use Cake\Core\Configure;
use Cake\Core\Configure\Engine\PhpConfig;

Configure::config('default', new PhpConfig());
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

Você também pode usar arquivos de configuração adicionais para fornecer sobreposições específicas do ambiente. Cada arquivo carregado após **app.php** pode redefinir valores previamente declarados permitindo que você personalize a configuração para ambientes de desenvolvimento ou de homologação.

Configuração Geral

Abaixo está uma descrição das variáveis e como elas afetam seu aplicativo CakePHP.

debug Altera a saída de depuração do CakePHP. `false` = Modo Produção. Não é exibido nenhuma mensagem de erro e/ou aviso. `true` = Modo de Desenvolvimento. É exibido todas as mensagens de erros e/ou avisos.

App.namespace O namespace em que as classes do aplicativo estão.

Nota: Ao alterar o namespace em sua configuração, você também precisará atualizar o arquivo `** composer.json **` para usar esse namespace também. Além disso, crie um novo carregador automático executando `php composer.phar dumpautoload`.

App.baseUrl Não comentar esta definição se você **não** planeja usar o `mod_rewrite` do Apache com o CakePHP. Não se esqueça de remover seus arquivos `.htaccess` também.

App.base O diretório base no qual o aplicativo reside. Se `false` isso será detectado automaticamente. Se não `false`, certifique-se de que sua sequência de caracteres começa com um `/` e **NÃO** termina com um `/`. Por exemplo, `/basedir` deve ser uma `App.base` válida. Caso contrário, o `AuthComponent` não funcionará corretamente.

App.encoding Defina a codificação que seu aplicativo usa. Essa codificação é usada para gerar o `charset` no layout e codificar entidades. Ele deve corresponder aos valores de codificação especificados para o seu banco de dados.

App.webroot O diretório raiz da aplicação web.

App.wwwRoot O diretório raiz dos arquivos da aplicação web.

App.fullBaseUrl O nome de domínio totalmente qualificado (incluindo o protocolo) para a raiz do aplicativo. Isso é usado ao gerar URLs absolutos. Por padrão, esse valor é gerado usando a variável `$_SERVER`. Entretanto, Você deve defini-lo manualmente para otimizar o desempenho ou se você está preocupado com as pessoas manipulando o cabeçalho do `Host`. Em um contexto CLI (do Shell) a `fullBaseUrl` não pode ser lido a partir de `$_SERVER`, como não há servidor envolvido. Você precisará especificá-lo se precisar gerar URLs de um shell (por exemplo, ao enviar e-mails).

App.imageBaseUrl O caminho da web para as imagens públicas na webroot da aplicação. Se você estiver usando um [CDN](#), você deve definir este valor para a localização do CDN.

App.cssBaseUrl O caminho da web para os arquivos de estilos em cascata(`.css`) públicos na webroot da aplicação. Se você estiver usando um [CDN](#), você deve definir este valor para a localização do CDN.

App.jsBaseUrl O caminho da web para os scripts (em JavaScript) públicos na webroot da aplicação. Se você estiver usando um [CDN](#), você deve definir este valor para a localização do CDN.

App.paths Configurar caminhos para recursos não baseados em classe. Suporta as subchaves `plugins`, `templates`, `locales`, que permitem a definição de caminhos para plugins, templates e arquivos de locale respectivamente.

Security.salt Uma sequência aleatória usada em hash. Uma sequência aleatória usada em hash. Este valor também é usado como o sal HMAC ao fazer criptografia simétrica.

Asset.timestamp Acrescenta um carimbo de data/hora que é a última hora modificada do arquivo específico no final dos URLs de arquivos de recurso (CSS, JavaScript, Image) ao usar assistentes adequados. Valores válidos:

- (bool) `false` - Não fazer nada (padrão)
- (bool) `true` - Acrescenta o carimbo de data/hora quando depuração é `true`
- (string) `'force'` - Sempre anexa o carimbo de data/hora.

Configuração do banco de dados

Consulte [Database Configuration](#) para obter informações sobre como configurar suas conexões de banco de dados.

Configuração do Cache

Consulte [Caching Configuration](#) para obter informações sobre como configurar o cache no CakePHP.

Configuração de manipulação de erro e exceção

Consulte [Error and Exception Configuration](#) para obter informações sobre como configurar manipuladores de erro e exceção.

Configuração de log

Consulte [Logging Configuration](#) para obter informações sobre como configurar o log no CakePHP.

Configuração de e-mail

Consulte [Email Configuration](#) para obter informações sobre como configurar predefinições de e-mail no CakePHP.

Configuração de sessão

Consulte [Session Configuration](#) para obter informações sobre como configurar o tratamento de sessão no CakePHP.

Configuração de roteamento

Consulte [Routes Configuration](#) para obter mais informações sobre como configurar o roteamento e criar rotas para seu aplicativo.

Caminhos adicionais de classe

Caminhos de classe adicionais são configurados através dos carregadores automáticos usados pelo aplicativo. Ao usar o Composer para gerar o seu arquivo de autoload, você pode fazer o seguinte, para fornecer caminhos alternativos para controladores em seu aplicativo:

```
"autoload": {
    "psr-4": {
        "App\\Controller\\": "/path/to/directory/with/controller/folders",
        "App\\": "src"
    }
}
```

O código acima seria configurar caminhos para o namespace App e App\\Controller. A primeira chave será pesquisada e, se esse caminho não contiver a classe/arquivo, a segunda chave será pesquisada. Você também pode mapear um namespace único para vários diretórios com o seguinte código:

```
"autoload": {
    "psr-4": {
        "App\\": ["src", "/path/to/directory"]
    }
}
```

Plugin, Modelos de Visualização e Caminhos Locais

Como os plug-ins, os modelos de visualização (Templates) e os caminhos locais (locales) não são classes, eles não podem ter um autoloader configurado. O CakePHP fornece três variáveis de configuração para configurar caminhos adicionais para esses recursos. No **config/app.php** você pode definir estas variáveis

```
return [
    // More configuration
    'App' => [
        'paths' => [
            'plugins' => [
                ROOT . DS . 'plugins' . DS,
                '/path/to/other/plugins/'
            ],
            'templates' => [
                APP . 'Template' . DS,
                APP . 'Template2' . DS
            ],
            'locales' => [
                APP . 'Locale' . DS
            ]
        ]
    ]
];
```

Caminhos devem terminar com um separador de diretório, ou eles não funcionarão corretamente.

Configuração de Inflexão

Consulte [Configuração da inflexão](#) para obter mais informações sobre como fazer a configuração de inflexão.

Configurar classe

class Cake\Core\Configure

A classe de Configuração do CakePHP pode ser usada para armazenar e recuperar valores específicos do aplicativo ou do tempo de execução. Tenha cuidado, pois essa classe permite que você armazene qualquer coisa nela, para que em seguida, usá-la em qualquer outra parte do seu código: Dando ma certa tentação de quebrar o padrão MVC do CakePHP. O objetivo principal da classe Configurar é manter variáveis centralizadas que podem ser compartilhadas entre muitos objetos. Lembre-se de tentar viver por “convenção sobre a configuração” e você não vai acabar quebrando a estrutura MVC previamente definida.

Você pode acessar o Configure de qualquer lugar de seu aplicativo:

```
Configure::read('debug');
```

Escrevendo dados de configuração

static Cake\Core\Configure::write(\$key, \$value)

Use write() para armazenar dados na configuração do aplicativo:

```
Configure::write('Company.name', 'Pizza, Inc.');
```

```
Configure::write('Company.slogan', 'Pizza for your body and soul');
```

Nota: O *dot notation* usado no parâmetro `$key` pode ser usado para organizar suas configurações em grupos lógicos.

O exemplo acima também pode ser escrito em uma única chamada:

```
Configure::write('Company', [
    'name' => 'Pizza, Inc.',
    'slogan' => 'Pizza for your body and soul'
]);
```

Você pode usar `Configure::write('debug', $bool)` para alternar entre os modos de depuração e produção na mosca. Isso é especialmente útil para interações JSON onde informações de depuração podem causar problemas de análise.

Leitura de dados de configuração

static `Cake\Core\Configure::read($key = null)`

Usado para ler dados de configuração da aplicação. Por padrão o valor de depuração do CakePHP é importante. Se for fornecida uma chave, os dados são retornados. Usando nossos exemplos de `write()` acima, podemos ler os dados de volta:

```
Configure::read('Company.name');    // Yields: 'Pizza, Inc.'
Configure::read('Company.slogan');  // Yields: 'Pizza for your body
                                   // and soul'

Configure::read('Company');
```

```
//Rendimentos:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];
```

Se `$key` for deixada nula, todos os valores em `Configure` serão retornados.

static `Cake\Core\Configure::readOrFail($key)`

Lê dados de configuração como `Cake\Core\Configure::read`, mas espera encontrar um par chave/valor. Caso o par solicitado não exista, a `RuntimeException` será lançada:

```
Configure::readOrFail('Company.name');    // Rendimentos: 'Pizza, Inc.'
Configure::readOrFail('Company.geolocation'); // Vai lançar uma exceção

Configure::readOrFail('Company');
```

```
// Rendimentos:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];
```

Novo na versão 3.1.7: `Configure::readOrFail()` Foi adicionado na versão 3.1.7

Verificar se os dados de configuração estão definidos

static `Cake\Core\Configure::check($key)`

Usado para verificar se uma chave/caminho existe e tem valor não nulo:

```
$exists = Configure::check('Company.name');
```

Excluindo Dados de Configuração

static Cake\Core\Configure::delete(\$key)

Usado para excluir informações da configuração da aplicação:

```
Configure::delete('Company.name');
```

Leitura e exclusão de dados de configuração

static Cake\Core\Configure::consume(\$key)

Ler e excluir uma chave do Configure. Isso é útil quando você deseja combinar leitura e exclusão de valores em uma única operação.

Lendo e escrevendo arquivos de configuração

static Cake\Core\Configure::config(\$name, \$engine)

O CakePHP vem com dois mecanismos de arquivos de configuração embutidos. `Cake\Core\Configure\Engine\PhpConfig` é capaz de ler arquivos de configuração do PHP, no mesmo formato que o Configure tem lido historicamente. `Cake\Core\Configure\Engine\IniConfig` é capaz de ler os arquivos de configuração no formato ini(.ini). Consulte a documentação do PHP⁷⁵ para obter mais informações sobre os detalhes dos arquivos ini. Para usar um mecanismo de configuração do núcleo, você precisará conectá-lo ao Configure usando `Configure::config()`:

```
use Cake\Core\Configure\Engine\PhpConfig;

// Ler os arquivos de configuração da configuração
Configure::config('default', new PhpConfig());

// Ler arquivos de configuração de outro diretório.
Configure::config('default', new PhpConfig('/path/to/your/config/files/'));
```

Você pode ter vários mecanismos anexados para Configure, cada um lendo diferentes tipos ou fontes de arquivos de configuração. Você pode interagir com os motores conectados usando alguns outros métodos em Configure. Para verificar quais aliases de motor estão conectados você pode usar `Configure::configured()`:

```
// Obter a matriz de aliases para os motores conectados.
Configure::configured();

// Verificar se um motor específico está ligado.
Configure::configured('default');
```

static Cake\Core\Configure::drop(\$name)

Você também pode remover os motores conectados. `Configure::drop('default')` removeria o alias de mecanismo padrão. Quaisquer tentativas futuras de carregar arquivos de configuração com esse mecanismo falhariam:

⁷⁵ http://php.net/parse_ini_file


```
Configure::drop('default');
```

Carregando arquivos de configurações

static `Cake\Core\Configure::load($key, $config = 'default', $merge = true)`

Depois de ter anexado um motor de configuração para o Configure, ficará disponível para poder carregar ficheiros de configuração:

```
// Load my_file.php using the 'default' engine object.
Configure::load('my_file', 'default');
```

Os arquivos de configuração que foram carregados mesclam seus dados com a configuração de tempo de execução existente no Configure. Isso permite que você sobrescreva e adicione novos valores à configuração de tempo de execução existente. Ao definir `$merge` para `true`, os valores nunca substituirão a configuração existente.

Criando ou modificando arquivos de configuração

static `Cake\Core\Configure::dump($key, $config = 'default', $keys = [])`

Despeja todos ou alguns dos dados que estão no Configure em um sistema de arquivos ou armazenamento suportado por um motor de configuração. O formato de serialização é decidido pelo mecanismo de configuração anexado como `$config`. Por exemplo, se o mecanismo 'padrão' é `Cake\Core\Configure\Engine\PhpConfig`, o arquivo gerado será um arquivo de configuração PHP carregável pelo `Cake\Core\Configure\Engine\PhpConfig`.

Dado que o motor 'default' é uma instância do `PhpConfig`. Salve todos os dados em Configure no arquivo `my_config.php`:

```
Configure::dump('my_config', 'default');
```

Salvar somente a configuração de manipulação de erro:

```
Configure::dump('error', 'default', ['Error', 'Exception']);
```

`Configure::dump()` pode ser usado para modificar ou substituir arquivos de configuração que são legíveis com `Configure::load()`.

Armazenando Configuração do Tempo de Execução

static `Cake\Core\Configure::store($name, $cacheConfig = 'default', $data = null)`

Você também pode armazenar valores de configuração de tempo de execução para uso em uma solicitação futura. Como o configure só lembra valores para a solicitação atual, você precisará armazenar qualquer informação de configuração modificada se você quiser usá-la em solicitações futuras:

```
// Armazena a configuração atual na chave 'user_1234' no cache 'default'.
Configure::store('user_1234', 'default');
```

Os dados de configuração armazenados são mantidos na configuração de cache nomeada. Consulte a documentação [Caching](#) para obter mais informações sobre o cache.

Restaurando a Configuração do Tempo de Execução

static `Cake\Core\Configure::restore($name, $cacheConfig = 'default')`

Depois de ter armazenado a configuração de tempo de execução, você provavelmente precisará restaurá-la para que você possa acessá-la novamente. `Configure::restore()` faz exatamente isso:

```
// Restaura a configuração do tempo de execução do cache.
Configure::restore('user_1234', 'default');
```

Ao restaurar informações de configuração, é importante restaurá-lo com a mesma chave e configuração de cache usada para armazená-lo. As informações restauradas são mescladas em cima da configuração de tempo de execução existente.

Criando seus próprios mecanismos de configuração

Como os mecanismos de configuração são uma parte extensível do CakePHP, você pode criar mecanismos de configuração em seu aplicativo e plugins. Os motores de configuração precisam de uma `Cake\Core\Configure\ConfigEngineInterface`. Esta interface define um método de leitura, como o único método necessário. Se você gosta de arquivos XML, você pode criar um motor de XML de configuração simples para sua aplicação:

```
// Em src/Configure/Engine/XmlConfig.php
namespace App\Configure\Engine;

use Cake\Core\Configure\ConfigEngineInterface;
use Cake\Utility\Xml;

class XmlConfig implements ConfigEngineInterface
{
    public function __construct($path = null)
    {
        if (!$path) {
            $path = CONFIG;
        }
        $this->_path = $path;
    }

    public function read($key)
    {
        $xml = Xml::build($this->_path . $key . '.xml');
        return Xml::toArray($xml);
    }

    public function dump($key, array $data)
    {
        // Code to dump data to file
    }
}
```

No seu `config/bootstrap.php` você poderia anexar este mecanismo e usá-lo:

```
use App\Configure\Engine\XmlConfig;

Configure::config('xml', new XmlConfig());
```

```
...
Configure::load('my_xml', 'xml');
```

O método `read()` de um mecanismo de configuração, deve retornar uma matriz das informações de configuração que o recurso chamado `$key` contém.

interface Cake\Core\Configure\ConfigEngineInterface

Define a interface usada pelas classes que lêem dados de configuração e armazenam-no em `Configure`

Cake\Core\Configure\ConfigEngineInterface::read(*\$key*)

Parâmetros

- **\$key** (*string*) – O nome da chave ou identificador a carregar.

Esse método deve carregar/analisar os dados de configuração identificados pelo `$key` e retornar uma matriz de dados no arquivo.

Cake\Core\Configure\ConfigEngineInterface::dump(*\$key*)

Parâmetros

- **\$key** (*string*) – O identificador para escrever.
- **\$data** (*array*) – Os dados para despejo.

Esse método deve despejar/armazenar os dados de configuração fornecidos para uma chave identificada pelo `$key`.

Motores de Configuração Integrados

Arquivos de configuração do PHP

class Cake\Core\Configure\Engine\PhpConfig

Permite ler arquivos de configuração que são armazenados como arquivos simples do PHP. Você pode ler arquivos da configuração do aplicativo ou do plugin configs diretórios usando *sintaxe plugin*. Arquivos *devem* retornar uma matriz. Um exemplo de arquivo de configuração seria semelhante a:

```
return [
    'debug' => 0,
    'Security' => [
        'salt' => 'its-secret'
    ],
    'App' => [
        'namespace' => 'App'
    ]
];
```

Carregue seu arquivo de configuração personalizado inserindo o seguinte em `config/bootstrap.php`:

```
Configure::load('customConfig');
```

Arquivos de configuração Ini

class Cake\Core\Configure\Engine\IniConfig

Permite ler arquivos de configuração armazenados como arquivos .ini simples. Os arquivos ini devem ser compatíveis com a função `parse_ini_file()` do php e beneficiar das seguintes melhorias.

- Os valores separados por ponto são expandidos em arrays.
- Valores booleanos como 'on' e 'off' são convertidos em booleanos.

Um exemplo de arquivo ini seria semelhante a:

```
debug = 0

[Security]
salt = its-secret

[App]
namespace = App
```

O arquivo ini acima, resultaria nos mesmos dados de configuração final do exemplo PHP acima. As estruturas de matriz podem ser criadas através de valores separados por pontos ou por seções. As seções podem conter chaves separadas por pontos para um assentamento mais profundo.

Arquivos de configuração do Json

class Cake\Core\Configure\Engine\JsonConfig

Permite ler/descarregar arquivos de configuração armazenados como cadeias codificadas JSON em arquivos .json.

Um exemplo de arquivo JSON seria semelhante a:

```
{
    "debug": false,
    "App": {
        "namespace": "MyApp"
    },
    "Security": {
        "salt": "its-secret"
    }
}
```

Bootstrapping CakePHP

Se você tiver alguma necessidade de configuração adicional, adicione-a ao arquivo **config/bootstrap.php** do seu aplicativo. Este arquivo é incluído antes de cada solicitação, e o comando CLI.

Este arquivo é ideal para várias tarefas de bootstrapping comuns:

- Definir funções de conveniência.
- Declaração de constantes.
- Definição da configuração do cache.
- Definição da configuração de log.
- Carregando inflexões personalizadas.
- Carregando arquivos de configuração.

Pode ser tentador para colocar as funções de formatação lá, a fim de usá-los em seus controladores. Como você verá nas seções *Controllers (Controladores)* e *Views (Visualização)* há melhores maneiras de adicionar lógica personalizada à sua aplicação.

Application::bootstrap()

Além do arquivo **config/bootstrap.php** que deve ser usado para configurar preocupações de baixo nível do seu aplicativo, você também pode usar o método `Application::bootstrap()` para carregar/inicializar plugins, E anexar ouvintes de eventos globais:

```
// Em src/Application.php
namespace App;

use Cake\Core\Plugin;
use Cake\Http\BaseApplication;

class Application extends BaseApplication
{
    public function bootstrap()
    {
        // Chamar o pai para `require_once` config/bootstrap.php
        parent::bootstrap();

        Plugin::load('MyPlugin', ['bootstrap' => true, 'routes' => true]);
    }
}
```

Carregar plugins/eventos em `Application::bootstrap()` torna *Controller Integration Testing* mais fácil à medida que os eventos e rotas serão re-processados em cada método de teste.

Variáveis de Ambiente

Alguns dos provedores modernos de nuvem, como o Heroku, permitem definir variáveis de ambiente. Ao definir variáveis de ambiente, você pode configurar seu aplicativo CakePHP como um aplicativo 12factor. Seguir as instruções do aplicativo [12factor app instructions](#)⁷⁶ é uma boa maneira de criar um app sem estado e facilitar a implantação do seu aplicativo. Isso significa, por exemplo, que, se você precisar alterar seu banco de dados, você precisará modificar uma variável DATABASE_URL na sua configuração de host sem a necessidade de alterá-la em seu código-fonte.

Como você pode ver no seu **app.php**, as seguintes variáveis estão em uso:

- DEBUG (0 ou "1")
- APP_ENCODING (ie UTF-8)
- APP_DEFAULT_LOCALE (ie en_US)
- SECURITY_SALT
- CACHE_DEFAULT_URL (ie File:///prefix=myapp_&serialize=true&timeout=3600&path=../tmp/cache)
- CACHE_CAKECORE_URL (ie File:///prefix=myapp_cake_core_&serialize=true&timeout=3600&path=../tmp/cache)
- CACHE_CAKEMODEL_URL (ie File:///prefix=myapp_cake_model_&serialize=true&timeout=3600&path=../tmp/cache)
- EMAIL_TRANSPORT_DEFAULT_URL (ie smtp://user:password@hostname:port?tls=null&client=null&transport=smtp)

⁷⁶ <http://12factor.net/>

- DATABASE_URL (ie mysql://user:pass@db/my_app)
- DATABASE_TEST_URL (ie mysql://user:pass@db/test_my_app)
- LOG_DEBUG_URL (ie file:///?levels[]=notice&levels[]=info&levels[]=debug&file=debug&path=.
- LOG_ERROR_URL (ie file:///?levels[]=warning&levels[]=error&levels[]=critical&levels[]=ale

Como você pode ver nos exemplos, definimos algumas opções de configuração como *DSN*. Este é o caso de bancos de dados, logs, transporte de e-mail e configurações de cache.

Se as variáveis de ambiente não estiverem definidas no seu ambiente, o CakePHP usará os valores definidos no **app.php**. Você pode usar a biblioteca [php-dotenv library](https://github.com/josegonzalez/php-dotenv)⁷⁷ para usar variáveis de ambiente em um desenvolvimento local. Consulte as instruções Leiamos da biblioteca para obter mais informações.

Desabilitando tabelas genéricas

Embora a utilização de classes de tabela genéricas - também chamadas auto-tables - quando a criação rápida de novos aplicativos e modelos de cozimento é útil, a classe de tabela genérica pode tornar a depuração mais difícil em alguns cenários.

Você pode verificar se qualquer consulta foi emitida de uma classe de tabela genérica via DebugKit através do painel SQL no DebugKit. Se você ainda tiver problemas para diagnosticar um problema que pode ser causado por tabelas automáticas, você pode lançar uma exceção quando o CakePHP implicitamente usa um Cake\ORM\Table genérico em vez de sua classe concreta assim:

```
// No seu bootstrap.php
use Cake\Event\EventManager;
use Cake\Network\Exception\InternalErrorException;

$isCakeBakeShellRunning = (PHP_SAPI === 'cli' && isset($argv[1]) && $argv[1] === 'bake
↪');
if (!$isCakeBakeShellRunning) {
    EventManager::instance()->on('Model.initialize', function($event) {
        $subject = $event->getSubject();
        if (get_class($subject) === 'Cake\ORM\Table') {
            $msg = sprintf(
                'Missing table class or incorrect alias when registering table class_
↪for database table %s.',
                $subject->getTable());
            throw new InternalErrorException($msg);
        }
    });
}
```

⁷⁷ <https://github.com/josegonzalez/php-dotenv>

Roteamento

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)⁷⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁷⁸ <https://github.com/cakephp/docs>

Routing

class Cake\Routing\Router

Routing provides you tools that map URLs to controller actions. By defining routes, you can separate how your application is implemented from how its URL's are structured.

Routing in CakePHP also encompasses the idea of reverse routing, where an array of parameters can be transformed into a URL string. By using reverse routing, you can re-factor your application's URL structure without having to update all your code.

Quick Tour

This section will teach you by example the most common uses of the CakePHP Router. Typically you want to display something as a landing page, so you add this to your **routes.php** file:

```
use Cake\Routing\Router;

// Using the scoped route builder.
Router::scope('/', function ($routes) {
    $routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
});

// Using the static method.
Router::connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Router provides two interfaces for connecting routes. The static method is a backwards compatible interface, while the scoped builders offer more terse syntax when building multiple routes, and better performance.

This will execute the index method in the ArticlesController when the homepage of your site is visited. Sometimes you need dynamic routes that will accept multiple parameters, this would be the case, for example of a route for viewing an article's content:

```
$routes->connect('/articles/*', ['controller' => 'Articles', 'action' => 'view']);
```

The above route will accept any URL looking like /articles/15 and invoke the method view(15) in the ArticlesController. This will not, though, prevent people from trying to access URLs looking like /articles/foobar. If you wish, you can restrict some parameters to conform to a regular expression:

```
$routes->connect(
    '/articles/:id',
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// Prior to 3.5 use the options array
$routes->connect(
    '/articles/:id',
    ['controller' => 'Articles', 'action' => 'view'],
    ['id' => '\d+', 'pass' => ['id']]
)
```

The previous example changed the star matcher by a new placeholder `:id`. Using placeholders allows us to validate parts of the URL, in this case we used the `\d+` regular expression so that only digits are matched. Finally, we told the Router to treat the `id` placeholder as a function argument to the `view()` function by specifying the `pass` option. More on using this option later.

The CakePHP Router can also reverse match routes. That means that from an array containing matching parameters, it is capable of generating a URL string:

```
use Cake\Routing\Router;

echo Router::url(['controller' => 'Articles', 'action' => 'view', 'id' => 15]);
// Will output
/articles/15
```

Routes can also be labelled with a unique name, this allows you to quickly reference them when building links instead of specifying each of the routing parameters:

```
// In routes.php
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'login']);
// Will output
/login
```

To help keep your routing code DRY, the Router has the concept of ‘scopes’. A scope defines a common path segment, and optionally route defaults. Any routes connected inside a scope will inherit the path/defaults from their wrapping scopes:

```
Router::scope('/blog', ['plugin' => 'Blog'], function ($routes) {
    $routes->connect('/', ['controller' => 'Articles']);
});
```

The above route would match `/blog/` and send it to `Blog\Controller\ArticlesController::index()`.

The application skeleton comes with a few routes to get you started. Once you’ve added your own routes, you can remove the default routes if you don’t need them.

Connecting Routes

`Cake\Routing\Router::connect ($route, $defaults = [], $options = [])`

To keep your code *DRY* you should use ‘routing scopes’. Routing scopes not only let you keep your code DRY, they also help Router optimize its operation. This method defaults to the `/` scope. To create a scope and connect some routes we’ll use the `scope()` method:

```
// In config/routes.php
use Cake\Routing\Route\DashedRoute;

Router::scope('/', function ($routes) {
    // Connect the generic fallback routes.
    $routes->fallbacks(DashedRoute::class);
});
```

The `connect()` method takes up to three parameters: the URL template you wish to match, the default values for your route elements, and the options for the route. Options frequently include regular expression rules to help the router match elements in the URL.

The basic format for a route definition is:

```
$routes->connect (
    '/url/template',
    ['default' => 'defaultValue'],
    ['option' => 'matchingRegex']
);
```

The first parameter is used to tell the router what sort of URL you’re trying to control. The URL is a normal slash delimited string, but can also contain a wildcard (*) or *Route Elements*. Using a wildcard tells the router that you are willing to accept any additional arguments supplied. Routes without a * only match the exact template pattern supplied.

Once you’ve specified a URL, you use the last two parameters of `connect()` to tell CakePHP what to do with a request once it has been matched. The second parameter is an associative array. The keys of the array should be named after the route elements the URL template represents. The values in the array are the default values for those keys. Let’s look at some basic examples before we start using the third parameter of `connect()`:

```
$routes->connect (
    '/pages/*',
    ['controller' => 'Pages', 'action' => 'display']
);
```

This route is found in the `routes.php` file distributed with CakePHP. It matches any URL starting with `/pages/` and hands it to the `display()` action of the `PagesController`. A request to `/pages/products` would be mapped to `PagesController->display('products')`.

In addition to the greedy star `/*` there is also the `/**` trailing star syntax. Using a trailing double star, will capture the remainder of a URL as a single passed argument. This is useful when you want to use an argument that included a `/` in it:

```
$routes->connect (
    '/pages/**',
    ['controller' => 'Pages', 'action' => 'show']
);
```

The incoming URL of `/pages/the-example-/and-proof` would result in a single passed argument of `the-example-/and-proof`.

You can use the second parameter of `connect()` to provide any routing parameters that are composed of the default values of the route:

```
$routes->connect (
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

This example shows how you can use the second parameter of `connect()` to define default parameters. If you built a site that features products for different categories of customers, you might consider creating a route. This allows you to link to `/government` rather than `/pages/display/5`.

A common use for routing is to create URL segments that don't match your controller or model names. Let's say that instead of accessing our regular URL at `/users/some_action/5`, we'd like to be able to access it by `/cooks/some_action/5`. The following route takes care of that:

```
$routes->connect (
    '/cooks/:action/*', ['controller' => 'Users']
);
```

This is telling the Router that any URL beginning with `/cooks/` should be sent to the `UsersController`. The action called will depend on the value of the `:action` parameter. By using *Route Elements*, you can create variable routes, that accept user input or variables. The above route also uses the greedy star. The greedy star indicates that this route should accept any additional positional arguments given. These arguments will be made available in the *Passed Arguments* array.

When generating URLs, routes are used too. Using `['controller' => 'Users', 'action' => 'some_action', 5]` as a URL will output `/cooks/some_action/5` if the above route is the first match found.

The routes we've connected so far will match any HTTP verb. If you are building a REST API you'll often want to map HTTP actions to different controller methods. The `RouteBuilder` provides helper methods that make defining routes for specific HTTP verbs simpler:

```
// Create a route that only responds to GET requests.
$routes->get (
    '/cooks/:id',
    ['controller' => 'Users', 'action' => 'view'],
    'users:view'
);

// Create a route that only responds to PUT requests
$routes->put (
    '/cooks/:id',
    ['controller' => 'Users', 'action' => 'update'],
    'users:update'
);
```

The above routes map the same URL to different controller actions based on the HTTP verb used. GET requests will go to the 'view' action, while PUT requests will go to the 'update' action. There are HTTP helper methods for:

- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS

- HEAD

All of these methods return the route instance allowing you to leverage the *fluent setters* to further configure your route.

Novo na versão 3.5.0: The HTTP verb helper methods were added in 3.5.0

Route Elements

You can specify your own route elements and doing so gives you the power to define places in the URL where parameters for controller actions should lie. When a request is made, the values for these route elements are found in `$this->request->getParam()` in the controller. When you define a custom route element, you can optionally specify a regular expression - this tells CakePHP how to know if the URL is correctly formed or not. If you choose to not provide a regular expression, any non / character will be treated as part of the parameter:

```
$routes->connect (
   ('/:controller/:id',
    ['action' => 'view']
)->setPatterns(['id' => '[0-9]+']);

// Prior to 3.5 use the options array
$routes->connect (
   ('/:controller/:id',
    ['action' => 'view'],
    ['id' => '[0-9]+']
);
```

The above example illustrates how to create a quick way to view models from any controller by crafting a URL that looks like `/controllername/:id`. The URL provided to `connect()` specifies two route elements: `:controller` and `:id`. The `:controller` element is a CakePHP default route element, so the router knows how to match and identify controller names in URLs. The `:id` element is a custom route element, and must be further clarified by specifying a matching regular expression in the third parameter of `connect()`.

CakePHP does not automatically produce lowercased and dashed URLs when using the `:controller` parameter. If you need this, the above example could be rewritten like so:

```
use Cake\Routing\Route\DashedRoute;

// Create a builder with a different route class.
$routes->scope('/', function ($routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/:controller/:id', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    // Prior to 3.5 use options array
    $routes->connect (
       ('/:controller/:id',
        ['action' => 'view'],
        ['id' => '[0-9]+']
    );
});
```

The `DashedRoute` class will make sure that the `:controller` and `:plugin` parameters are correctly lowercased and dashed.

If you need lowercased and underscored URLs while migrating from a CakePHP 2.x application, you can instead use the `InflectedRoute` class.

Nota: Patterns used for route elements must not contain any capturing groups. If they do, Router will not function correctly.

Once this route has been defined, requesting `/apples/5` would call the `view()` method of the `ApplesController`. Inside the `view()` method, you would need to access the passed ID at `$this->request->getParam('id')`.

If you have a single controller in your application and you do not want the controller name to appear in the URL, you can map all URLs to actions in your controller. For example, to map all URLs to actions of the `home` controller, e.g have URLs like `/demo` instead of `/home/demo`, you can do the following:

```
$routes->connect('/:action', ['controller' => 'Home']);
```

If you would like to provide a case insensitive URL, you can use regular expression inline modifiers:

```
// Prior to 3.5 use the options array instead of setPatterns()
$routes->connect (
    '/:userShortcut',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

One more example, and you'll be a routing pro:

```
// Prior to 3.5 use the options array instead of setPatterns()
$routes->connect (
    '/:controller/:year/:month/:day',
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|12[0-9]|3[01]'
]);
```

This is rather involved, but shows how powerful routes can be. The URL supplied has four route elements. The first is familiar to us: it's a default route element that tells CakePHP to expect a controller name.

Next, we specify some default values. Regardless of the controller, we want the `index()` action to be called.

Finally, we specify some regular expressions that will match years, months and days in numerical form. Note that parenthesis (grouping) are not supported in the regular expressions. You can still specify alternates, as above, but not grouped with parenthesis.

Once defined, this route will match `/articles/2007/02/01`, `/articles/2004/11/16`, handing the requests to the `index()` actions of their respective controllers, with the date parameters in `$this->request->getParam()`.

There are several route elements that have special meaning in CakePHP, and should not be used unless you want the special meaning

- `controller` Used to name the controller for a route.
- `action` Used to name the controller action for a route.
- `plugin` Used to name the plugin a controller is located in.
- `prefix` Used for *Prefix Routing*
- `_ext` Used for *File extensions routing*.
- `_base` Set to `false` to remove the base path from the generated URL. If your application is not in the root directory, this can be used to generate URLs that are 'cake relative'.

- `_scheme` Set to create links on different schemes like *webcal* or *ftp*. Defaults to the current scheme.
- `_host` Set the host to use for the link. Defaults to the current host.
- `_port` Set the port if you need to create links on non-standard ports.
- `_full` If `true` the `FULL_BASE_URL` constant will be prepended to generated URLs.
- `#` Allows you to set URL hash fragments.
- `_ssl` Set to `true` to convert the generated URL to https or `false` to force http.
- `_method` Define the HTTP verb/method to use. Useful when working with *Criando rotas RESTful*.
- `_name` Name of route. If you have setup named routes, you can use this key to specify it.

Configuring Route Options

There are a number of route options that can be set on each route. After connecting a route you can use its fluent builder methods to further configure the route. These methods replace many of the keys in the `$options` parameter of `connect()`:

```
$routes->connect (
   ('/:lang/articles/:slug',
    ['controller' => 'Articles', 'action' => 'view'],
)
// Allow GET and POST requests.
->setMethods(['GET', 'POST'])

// Only match on the blog subdomain.
->setHost('blog.example.com')

// Set the route elements that should be converted to passed arguments
->setPass(['slug'])

// Set the matching patterns for route elements
->setPatterns([
    'slug' => '[a-z0-9-_-]+',
    'lang' => 'en|fr|es',
])

// Also allow JSON file extensions
->setExtensions(['json'])

// Set lang to be a persistent parameter
->setPersist(['lang']);
```

Novo na versão 3.5.0: Fluent builder methods were added in 3.5.0

Passing Parameters to Action

When connecting routes using *Route Elements* you may want to have routed elements be passed arguments instead. The `pass` option whitelists which route elements should also be made available as arguments passed into the controller functions:

```
// src/Controller/BlogsController.php
public function view($articleId = null, $slug = null)
{
```

```
// Some code here...
}

// routes.php
Router::scope('/', function ($routes) {
    $routes->connect(
        '/blog/:id-slug', // E.g. /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Define the route elements in the route template
    // to pass as function arguments. Order matters since this
    // will simply map ":id" to $articleId in your action
    ->setPass(['id', 'slug'])
    // Define a pattern that `id` must match.
    ->setPatterns([
        'id' => '[0-9]+',
    ]);
});
```

Now thanks to the reverse routing capabilities, you can pass in the URL array like below and CakePHP will know how to form the URL as defined in the routes:

```
// view.ctp
// This will return a link to /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
]);

// You can also use numerically indexed parameters.
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    3,
    'CakePHP_Rocks'
]);
```

Using Named Routes

Sometimes you'll find typing out all the URL parameters for a route too verbose, or you'd like to take advantage of the performance improvements that named routes have. When connecting routes you can specify a `__name` option, this option can be used in reverse routing to identify the route you want to use:

```
// Connect a route with a name.
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['__name' => 'login']
);

// Name a verb specific route (3.5.0+)
$routes->post(
    '/logout',
    ['controller' => 'Users', 'action' => 'logout'],
```



```

    'logout'
);

// Generate a URL using a named route.
$url = Router::url(['_name' => 'logout']);

// Generate a URL using a named route,
// with some query string args.
$url = Router::url(['_name' => 'login', 'username' => 'jimmy']);

```

If your route template contains any route elements like `:controller` you'll need to supply those as part of the options to `Router::url()`.

Nota: Route names must be unique across your entire application. The same `_name` cannot be used twice, even if the names occur inside a different routing scope.

When building named routes, you will probably want to stick to some conventions for the route names. CakePHP makes building up route names easier by allowing you to define name prefixes in each scope:

```

Router::scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
    // This route's name will be `api:ping`
    $routes->get('/ping', ['controller' => 'Pings', 'ping']);
});
// Generate a URL for the ping route
Router::url(['_name' => 'api:ping']);

// Use namePrefix with plugin()
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    // Connect routes.
});

// Or with prefix()
Router::prefix('Admin', ['_namePrefix' => 'admin:'], function ($routes) {
    // Connect routes.
});

```

You can also use the `_namePrefix` option inside nested scopes and it works as you'd expect:

```

Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
        // This route's name will be `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings', 'ping']);
    });
});

// Generate a URL for the ping route
Router::url(['_name' => 'contacts:api:ping']);

```

Routes connected in named scopes will only have names added if the route is also named. Nameless routes will not have the `_namePrefix` applied to them.

Novo na versão 3.1: The `_namePrefix` option was added in 3.1

Prefix Routing

static Cake\Routing\Router::**prefix**(\$name, \$callback)

Many applications require an administration section where privileged users can make changes. This is often done through a special URL such as /admin/users/edit/5. In CakePHP, prefix routing can be enabled by using the prefix scope method:

```
use Cake\Routing\Route\DashedRoute;

Router::prefix('admin', function ($routes) {
    // All routes here will be prefixed with `/admin`
    // And have the prefix => admin route element added.
    $routes->fallbacks(DashedRoute::class);
});
```

Prefixes are mapped to sub-namespaces in your application's Controller namespace. By having prefixes as separate controllers you can create smaller and simpler controllers. Behavior that is common to the prefixed and non-prefixed controllers can be encapsulated using inheritance, *Components (Componentes)*, or traits. Using our users example, accessing the URL /admin/users/edit/5 would call the edit() method of our **src/Controller/Admin/UsersController.php** passing 5 as the first parameter. The view file used would be **src/Template/Admin/Users/edit.ctp**

You can map the URL /admin to your index() action of pages controller using following route:

```
Router::prefix('admin', function ($routes) {
    // Because you are in the admin scope,
    // you do not need to include the /admin prefix
    // or the admin route element.
    $routes->connect('/', ['controller' => 'Pages', 'action' => 'index']);
});
```

When creating prefix routes, you can set additional route parameters using the \$options argument:

```
Router::prefix('admin', ['param' => 'value'], function ($routes) {
    // Routes connected here are prefixed with '/admin' and
    // have the 'param' routing key set.
    $routes->connect('/:controller');
});
```

You can define prefixes inside plugin scopes as well:

```
Router::plugin('DebugKit', function ($routes) {
    $routes->prefix('admin', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

The above would create a route template like /debug_kit/admin/:controller. The connected route would have the plugin and prefix route elements set.

When defining prefixes, you can nest multiple prefixes if necessary:

```
Router::prefix('manager', function ($routes) {
    $routes->prefix('admin', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

The above would create a route template like `/manager/admin/:controller`. The connected route would have the prefix route element set to `manager/admin`.

The current prefix will be available from the controller methods through `$this->request->getParam('prefix')`

When using prefix routes it's important to set the prefix option. Here's how to build this link using the HTML helper:

```
// Go into a prefixed route.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'manager', 'controller' => 'Articles', 'action' => 'add']
);

// Leave a prefix
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Nota: You should connect prefix routes *before* you connect fallback routes.

Plugin Routing

static `Cake\Routing\Router::plugin($name, $options = [], $callback)`

Routes for *Plugins* should be created using the `plugin()` method. This method creates a new routing scope for the plugin's routes:

```
Router::plugin('DebugKit', function ($routes) {
    // Routes connected here are prefixed with '/debug_kit' and
    // have the plugin route element set to 'DebugKit'.
    $routes->connect('/:controller');
});
```

When creating plugin scopes, you can customize the path element used with the `path` option:

```
Router::plugin('DebugKit', ['path' => '/debugger'], function ($routes) {
    // Routes connected here are prefixed with '/debugger' and
    // have the plugin route element set to 'DebugKit'.
    $routes->connect('/:controller');
});
```

When using scopes you can nest plugin scopes within prefix scopes:

```
Router::prefix('admin', function ($routes) {
    $routes->plugin('DebugKit', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

The above would create a route that looks like `/admin/debug_kit/:controller`. It would have the prefix, and plugin route elements set. The *Rotas para Plugin* section has more information on building plugin routes.

Creating Links to Plugin Routes

You can create links that point to a plugin, by adding the plugin key to your URL array:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => 'Todo', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Conversely if the active request is a plugin request and you want to create a link that has no plugin you can do the following:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => null, 'controller' => 'Users', 'action' => 'profile']
);
```

By setting 'plugin' => null you tell the Router that you want to create a link that is not part of a plugin.

SEO-Friendly Routing

Some developers prefer to use dashes in URLs, as it's perceived to give better search engine rankings. The `DashedRoute` class can be used in your application with the ability to route plugin, controller, and camelized action names to a dashed URL.

For example, if we had a `ToDo` plugin, with a `TodoItems` controller, and a `showItems()` action, it could be accessed at `/to-do/todo-items/show-items` with the following router connection:

```
use Cake\Routing\Route\DashedRoute;

Router::plugin('ToDo', ['path' => 'to-do'], function ($routes) {
    $routes->fallbacks(DashedRoute::class);
});
```

Matching Specific HTTP Methods

Routes can match specific HTTP methods using the HTTP verb helper methods:

```
Router::scope('/', function($routes) {
    // This route only matches on POST requests.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Match multiple verbs
    // Prior to 3.5 use $options['_method'] to set method
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
    )->setMethods(['POST', 'PUT']);
});
```

You can match multiple HTTP methods by using an array. Because the `_method` parameter is a routing key, it participates in both URL parsing and URL generation. To generate URLs for method specific routes you'll need to include the `_method` key when generating the URL:

```
$url = Router::url([
    'controller' => 'Reviews',
    'action' => 'start',
    '_method' => 'POST',
]);
```

Matching Specific Hostnames

Routes can use the `_host` option to only match specific hosts. You can use the `*` wildcard to match any subdomain:

```
Router::scope('/', function($routes) {
    // This route only matches on http://images.example.com
    // Prior to 3.5 use the _host option
    $routes->connect(
        '/images/default-logo.png',
        ['controller' => 'Images', 'action' => 'default']
    )->setHost('images.example.com');

    // This route only matches on http://*.example.com
    $routes->connect(
        '/images/old-log.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('images.example.com');
});
```

The `_host` option is also used in URL generation. If your `_host` option specifies an exact domain, that domain will be included in the generated URL. However, if you use a wildcard, then you will need to provide the `_host` parameter when generating URLs:

```
// If you have this route
$routes->connect(
    '/images/old-log.png',
    ['controller' => 'Images', 'action' => 'oldLogo']
)->setHost('images.example.com');

// You need this to generate a url
echo Router::url([
    'controller' => 'Images',
    'action' => 'oldLogo',
    '_host' => 'images.example.com',
]);
```

Novo na versão 3.4.0: The `_host` option was added in 3.4.0

Routing File Extensions

static `Cake\Routing\Router::extensions` (*string|array|null \$extensions, \$merge = true*)

To handle different file extensions with your routes, you can define extensions on a global, as well as on a scoped level. Defining global extensions can be achieved via the routers static `Router::extensions()` method:

```
Router::extensions(['json', 'xml']);
// ...
```

This will affect **all** routes that are being connected **afterwards**, no matter their scope.

In order to restrict extensions to specific scopes, you can define them using the `Cake\Routing\RouteBuilder::setExtensions()` method:

```
Router::scope('/', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $routes->setExtensions(['json', 'xml']);
});
```

This will enable the named extensions for all routes that are being connected in that scope **after** the `setExtensions()` call, including those that are being connected in nested scopes. Similar to the global `Router::extensions()` method, any routes connected prior to the call will not inherit the extensions.

Nota: Setting the extensions should be the first thing you do in a scope, as the extensions will only be applied to routes connected **after** the extensions are set.

Also be aware that re-opened scopes will **not** inherit extensions defined in previously opened scopes.

By using extensions, you tell the router to remove any matching file extensions, and then parse what remains. If you want to create a URL such as `/page/title-of-page.html` you would create your route using:

```
Router::scope('/page', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $routes->setExtensions(['json', 'xml', 'html']);
    $routes->connect(
        '/:title',
        ['controller' => 'Pages', 'action' => 'view']
    )->setPass(['title']);
});
```

Then to create links which map back to the routes simply use:

```
$this->Html->link(
    'Link title',
    ['controller' => 'Pages', 'action' => 'view', 'title' => 'super-article', '_ext' => 'html']
    => 'html'
);
```

File extensions are used by *Request Handling* to do automatic view switching based on content types.

Connecting Scoped Middleware

While Middleware can be applied to your entire application, applying middleware to specific routing scopes offers more flexibility, as you can apply middleware only where it is needed allowing your middleware to not concern itself with how/where it is being applied.

Before middleware can be applied to a scope, it needs to be registered into the route collection:

```
// in config/routes.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;
use Cake\Http\Middleware\EncryptedCookieMiddleware;
```

```
Router::scope('/', function ($routes) {
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
    $routes->registerMiddleware('cookies', new EncryptedCookieMiddleware());
});
```

Once registered, scoped middleware can be applied to specific scopes:

```
$routes->scope('/cms', function ($routes) {
    // Enable CSRF & cookies middleware
    $routes->applyMiddleware('csrf', 'cookies');
    $routes->get('/articles/:action/*', ['controller' => 'Articles'])
});
```

In situations where you have nested scopes, inner scopes will inherit the middleware applied in the containing scope:

```
$routes->scope('/api', function ($routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function ($routes) {
        $routes->applyMiddleware('v1compat');
        // Define routes here.
    });
});
```

In the above example, the routes defined in `/v1` will have `'ratelimit'`, `'auth.api'`, and `'v1compat'` middleware applied. If you re-open a scope, the middleware applied to routes in each scope will be isolated:

```
$routes->scope('/blog', function ($routes) {
    $routes->applyMiddleware('auth');
    // Connect the authenticated actions for the blog here.
});
$routes->scope('/blog', function ($routes) {
    // Connect the public actions for the blog here.
});
```

In the above example, the two uses of the `/blog` scope do not share middleware. However, both of these scopes will inherit middleware defined in their enclosing scopes.

Grouping Middleware

To help keep your route code DRY (Do not Repeat Yourself) middleware can be combined into groups. Once combined groups can be applied like middleware can:

```
$routes->registerMiddleware('cookie', new EncryptedCookieMiddleware());
$routes->registerMiddleware('auth', new AuthenticationMiddleware());
$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->middlewareGroup('web', ['cookie', 'auth', 'csrf']);

// Apply the group
$routes->applyMiddleware('web');
```

Novo na versão 3.5.0: Scoped middleware & middleware groups were added in 3.5.0

Criando rotas RESTful

O controle de rotas facilita a geração de rotas RESTful para seus controllers. Repousante as rotas são úteis quando você está criando pontos finais da API para sua aplicação. E se quermos permitir acesso REST a um controlador de receita, faríamos algo como esta:

```
// no arquivo config/routes.php...

Router::scope('/', function ($routes) {
    // anterior versao 3.5.0 usar `extensions()`
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

A primeira linha configura uma série de rotas padrão para REST fácil acesso onde o método especifica o formato de resultado desejado (por exemplo, xml,json, rss). Essas rotas são sensíveis ao método de solicitação HTTP.

HTTP format	URL.format	Controller action invoked
GET	/recipes.format	RecipesController::index()
GET	/recipes/123.format	RecipesController::view(123)
POST	/recipes.format	RecipesController::add()
PUT	/recipes/123.format	RecipesController::edit(123)
PATCH	/recipes/123.format	RecipesController::edit(123)
DELETE	/recipes/123.format	RecipesController::delete(123)

A classe CakePHP Router usa uma série de indicadores diferentes para detectar o método HTTP que está sendo usado. Aqui estão em ordem de preferência:

1. O `_method` POST variable
2. O `X_HTTP_METHOD_OVERRIDE`
3. O `REQUEST_METHOD` header

O `_method` POST variável é útil na utilização de um navegador como um cliente REST (ou qualquer outra coisa que possa fazer POST). Basta definir o valor do `_method` para o nome do método de solicitação HTTP que você deseja emular.

Creating Nested Resource Routes

Once you have connected resources in a scope, you can connect routes for sub-resources as well. Sub-resource routes will be prepended by the original resource name and a id parameter. For example:

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments');
    });
});
```

Will generate resource routes for both `articles` and `comments`. The comments routes will look like:

```
/api/articles/:article_id/comments
/api/articles/:article_id/comments/:id
```

You can get the `article_id` in `CommentsController` by:


```
$this->request->getParam('article_id');
```

By default resource routes map to the same prefix as the containing scope. If you have both nested and non-nested resource controllers you can use a different controller in each context by using prefixes:

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments', ['prefix' => 'articles']);
    });
});
```

The above would map the ‘Comments’ resource to the `App\Controller\Articles\CommentsController`. Having separate controllers lets you keep your controller logic simpler. The prefixes created this way are compatible with *Prefix Routing*.

Nota: While you can nest resources as deeply as you require, it is not recommended to nest more than 2 resources together.

Novo na versão 3.3: The `prefix` option was added to `resources()` in 3.3.

Limiting the Routes Created

By default CakePHP will connect 6 routes for each resource. If you’d like to only connect specific resource routes you can use the `only` option:

```
$routes->resources('Articles', [
    'only' => ['index', 'view']
]);
```

Would create read only resource routes. The route names are `create`, `update`, `view`, `index`, and `delete`.

Changing the Controller Actions Used

You may need to change the controller action names that are used when connecting routes. For example, if your `edit()` action is called `put()` you can use the `actions` key to rename the actions used:

```
$routes->resources('Articles', [
    'actions' => ['update' => 'put', 'create' => 'add']
]);
```

The above would use `put()` for the `edit()` action, and `add()` instead of `create()`.

Mapping Additional Resource Routes

You can map additional resource methods using the `map` option:

```
$routes->resources('Articles', [
    'map' => [
        'deleteAll' => [
            'action' => 'deleteAll',
            'method' => 'DELETE'
        ]
    ]
]);
```

```
    ]
  });
  // This would connect /articles/deleteAll
```

In addition to the default routes, this would also connect a route for `/articles/delete_all`. By default the path segment will match the key name. You can use the `'path'` key inside the resource definition to customize the path name:

```
$routes->resources('Articles', [
    'map' => [
        'updateAll' => [
            'action' => 'updateAll',
            'method' => 'DELETE',
            'path' => '/update_many'
        ],
    ],
]);
// This would connect /articles/update_many
```

If you define `'only'` and `'map'`, make sure that your mapped methods are also in the `'only'` list.

Custom Route Classes for Resource Routes

You can provide `connectOptions` key in the `$options` array for `resources()` to provide custom setting used by `connect()`:

```
Router::scope('/', function ($routes) {
    $routes->resources('Books', [
        'connectOptions' => [
            'routeClass' => 'ApiRoute',
        ],
    ]);
});
```

URL Inflection for Resource Routes

By default, multi-worded controllers' URL fragments are the underscored form of the controller's name. E.g., `BlogPostsController`'s URL fragment would be `/blog_posts`.

You can specify an alternative inflection type using the `inflect` option:

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'dasherize' // Will use ``Inflector::dasherize()``
    ]);
});
```

The above will generate URLs styled like: `/blog-posts`.

Nota: As of CakePHP 3.1 the official app skeleton uses `DashedRoute` as its default route class. Using the `'inflect' => 'dasherize'` option when connecting resource routes is recommended for URL consistency.

Changing the Path Element

By default resource routes use an inflected form of the resource name for the URL segment. You can set a custom URL segment with the `path` option:

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', ['path' => 'posts']);
});
```

Novo na versão 3.5.0: The `path` option was added in 3.5.0

Passed Arguments

Passed arguments are additional arguments or path segments that are used when making a request. They are often used to pass parameters to your controller methods.

```
http://localhost/calendars/view/recent/mark
```

In the above example, both `recent` and `mark` are passed arguments to `CalendarsController::view()`. Passed arguments are given to your controllers in three ways. First as arguments to the action method called, and secondly they are available in `$this->request->getParam('pass')` as a numerically indexed array. When using custom routes you can force particular parameters to go into the passed arguments as well.

If you were to visit the previously mentioned URL, and you had a controller action that looked like:

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
        debug(func_get_args());
    }
}
```

You would get the following output:

```
Array
(
    [0] => recent
    [1] => mark
)
```

This same data is also available at `$this->request->getParam('pass')` in your controllers, views, and helpers. The values in the `pass` array are numerically indexed based on the order they appear in the called URL:

```
debug($this->request->getParam('pass'));
```

Either of the above would output:

```
Array
(
    [0] => recent
    [1] => mark
)
```

When generating URLs, using a *routing array* you add passed arguments as values without string keys in the array:

```
['controller' => 'Articles', 'action' => 'view', 5]
```

Since 5 has a numeric key, it is treated as a passed argument.

Generating URLs

static Cake\Routing\Router::url (\$url = null, \$full = false)

Generating URLs or Reverse routing is a feature in CakePHP that is used to allow you to change your URL structure without having to modify all your code. By using *routing arrays* to define your URLs, you can later configure routes and the generated URLs will automatically update.

If you create URLs using strings like:

```
$this->Html->link('View', '/articles/view/' . $id);
```

And then later decide that /articles should really be called ‘posts’ instead, you would have to go through your entire application renaming URLs. However, if you defined your link like:

```
$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

Then when you decided to change your URLs, you could do so by defining a route. This would change both the incoming URL mapping, as well as the generated URLs.

When using array URLs, you can define both query string parameters and document fragments using special keys:

```
Router::url([
    'controller' => 'Articles',
    'action' => 'index',
    '?' => ['page' => 1],
    '#' => 'top'
]);

// Will generate a URL like.
/articles/index?page=1#top
```

Router will also convert any unknown parameters in a routing array to querystring parameters. The ? is offered for backwards compatibility with older versions of CakePHP.

You can also use any of the special route elements when generating URLs:

- `_ext` Used for *Routing File Extensions* routing.
- `_base` Set to `false` to remove the base path from the generated URL. If your application is not in the root directory, this can be used to generate URLs that are ‘cake relative’.
- `_scheme` Set to create links on different schemes like `webcal` or `ftp`. Defaults to the current scheme.
- `_host` Set the host to use for the link. Defaults to the current host.
- `_port` Set the port if you need to create links on non-standard ports.
- `_method` Define the HTTP verb the URL is for.
- `_full` If `true` the `FULL_BASE_URL` constant will be prepended to generated URLs.
- `_ssl` Set to `true` to convert the generated URL to `https` or `false` to force `http`.

- `_name` Name of route. If you have setup named routes, you can use this key to specify it.

Redirect Routing

Redirect routing allows you to issue HTTP status 30x redirects for incoming routes, and point them at different URLs. This is useful when you want to inform client applications that a resource has moved and you don't want to expose two URLs for the same content.

Redirection routes are different from normal routes as they perform an actual header redirection if a match is found. The redirection can occur to a destination within your application or an outside location:

```
Router::scope('/', function ($routes) {
    $routes->redirect(
        '/home/*',
        ['controller' => 'Articles', 'action' => 'view'],
        ['persist' => true]
        // Or ['persist'=>['id']] for default routing where the
        // view action expects $id as an argument.
    );
});
```

Redirects `/home/*` to `/articles/view` and passes the parameters to `/articles/view`. Using an array as the redirect destination allows you to use other routes to define where a URL string should be redirected to. You can redirect to external locations using string URLs as the destination:

```
Router::scope('/', function ($routes) {
    $routes->redirect('/articles/*', 'http://google.com', ['status' => 302]);
});
```

This would redirect `/articles/*` to `http://google.com` with a HTTP status of 302.

Custom Route Classes

Custom route classes allow you to extend and change how individual routes parse requests and handle reverse routing. Route classes have a few conventions:

- Route classes are expected to be found in the `Routing\Route` namespace of your application or plugin.
- Route classes should extend `Cake\Routing\Route`.
- Route classes should implement one or both of `match()` and/or `parse()`.

The `parse()` method is used to parse an incoming URL. It should generate an array of request parameters that can be resolved into a controller & action. Return `false` from this method to indicate a match failure.

The `match()` method is used to match an array of URL parameters and create a string URL. If the URL parameters do not match the route `false` should be returned.

You can use a custom route class when making a route by using the `routeClass` option:

```
$routes->connect(
   ('/:slug',
    ['controller' => 'Articles', 'action' => 'view'],
    ['routeClass' => 'SlugRoute']
);
```

```
// Or by setting the routeClass in your scope.
$routes->scope('/', function ($routes) {
    //Prior to 3.5.0 use `routeClass()`
    $routes->setRouteClass('SlugRoute');
    $routes->connect(
       ('/:slug',
        ['controller' => 'Articles', 'action' => 'view']
    );
});
```

This route would create an instance of `SlugRoute` and allow you to implement custom parameter handling. You can use plugin route classes using standard *sintaxe plugin*.

Default Route Class

static `Cake\Routing\Router::defaultRouteClass ($routeClass = null)`

If you want to use an alternate route class for all your routes besides the default `Route`, you can do so by calling `Router::defaultRouteClass()` before setting up any routes and avoid having to specify the `routeClass` option for each route. For example using:

```
use Cake\Routing\Route\InflectedRoute;

Router::defaultRouteClass(InflectedRoute::class);
```

will cause all routes connected after this to use the `InflectedRoute` route class. Calling the method without an argument will return current default route class.

Fallbacks Method

`Cake\Routing\Router::fallbacks ($routeClass = null)`

The fallbacks method is a simple shortcut for defining default routes. The method uses the passed routing class for the defined rules or if no class is provided the class returned by `Router::defaultRouteClass()` is used.

Calling fallbacks like so:

```
use Cake\Routing\Route\DashedRoute;

$routes->fallbacks(DashedRoute::class);
```

Is equivalent to the following explicit calls:

```
use Cake\Routing\Route\DashedRoute;

$routes->connect('/:controller', ['action' => 'index'], ['routeClass' => DashedRoute::
    ↪class]);
$routes->connect('/:controller/:action/*', [], ['routeClass' => DashedRoute::class]);
```

Nota: Using the default route class (`Route`) with fallbacks, or any route with `:plugin` and/or `:controller` route elements will result in inconsistent URL case.

Creating Persistent URL Parameters

You can hook into the URL generation process using URL filter functions. Filter functions are called *before* the URLs are matched against the routes, this allows you to prepare URLs before routing.

Callback filter functions should expect the following parameters:

- `$params` The URL params being processed.
- `$request` The current request.

The URL filter function should *always* return the params even if unmodified.

URL filters allow you to implement features like persistent parameters:

```
Router::addUrlFilter(function ($params, $request) {
    if ($request->getParam('lang') && !isset($params['lang'])) {
        $params['lang'] = $request->getParam('lang');
    }
    return $params;
});
```

Filter functions are applied in the order they are connected.

Another use case is changing a certain route on runtime (plugin routes for example):

```
Router::addUrlFilter(function ($params, $request) {
    if (empty($params['plugin']) || $params['plugin'] !== 'MyPlugin' || empty($params[
    ↪ 'controller'])) {
        return $params;
    }
    if ($params['controller'] === 'Languages' && $params['action'] === 'view') {
        $params['controller'] = 'Locations';
        $params['action'] = 'index';
        $params['language'] = $params[0];
        unset($params[0]);
    }
    return $params;
});
```

This will alter the following route:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Languages', 'action' => 'view',
    ↪ 'es']);
```

into this:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Locations', 'action' => 'index
    ↪', 'language' => 'es']);
```

Handling Named Parameters in URLs

Although named parameters were removed in CakePHP 3.0, applications may have published URLs containing them. You can continue to accept URLs containing named parameters.

In your controller's `beforeFilter()` method you can call `parseNamedParams()` to extract any named parameters from the passed arguments:

```
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
    Router::parseNamedParams($this->request);
}
```

This will populate `$this->request->getParam('named')` with any named parameters found in the passed arguments. Any passed argument that was interpreted as a named parameter, will be removed from the list of passed arguments.

Filtros do Dispatcher

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁷⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁷⁹ <https://github.com/cakephp/docs>

Objetos de requisição e resposta

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)⁸⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Requisição

Enviando Arquivos

⁸⁰ <https://github.com/cakephp/docs>

Controllers (Controladores)

class Cake\Controller\Controller

Os controllers (controladores) correspondem ao ‘C’ no padrão MVC. Após o roteamento ter sido aplicado e o controller correto encontrado, a ação do controller é chamada. Seu controller deve lidar com a interpretação dos dados de uma requisição, certificando-se que os models corretos são chamados e a resposta ou view esperada seja exibida. Os controllers podem ser vistos como intermediários entre a camada Model e View. Você vai querer manter seus controllers magros e seus Models gordos. Isso lhe ajudará a reutilizar seu código e testá-los mais facilmente.

Mais comumente, controllers são usados para gerenciar a lógica de um único model. Por exemplo, se você está construindo um site para uma padaria online, você pode ter um `RecipesController` e um `IngredientsController` gerenciando suas receitas e seus ingredientes. No CakePHP, controllers são nomeados de acordo com o model que manipulam. É também absolutamente possível ter controllers que usam mais de um model.

Os controllers da sua aplicação são classes que estendem a classe `AppController`, a qual por sua vez estende a classe do core `Controller`. A classe `AppController` pode ser definida em `src/Controller/AppController.php` e deve conter métodos que são compartilhados entre todos os controllers de sua aplicação.

Os controllers fornecem uma série de métodos que lidam com requisições. Estas são chamados de *actions*. Por padrão, todos os métodos públicos em um controller são uma action e acessíveis por uma URL. Uma action é responsável por interpretar a requisição e criar a resposta. Normalmente as respostas são na forma de uma view renderizada, mas também existem outras formas de criar respostas.

O App Controller

Como mencionado anteriormente, a classe `AppController` é a mãe de todos os outros controllers da sua aplicação. A própria `AppController` é estendida da classe `Cake\Controller\Controller` incluída no CakePHP. Assim sendo, `AppController` é definida em `src/Controller/AppController.php` como a seguir:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
}
```

Os atributos e métodos criados em seu `AppController` vão estar disponíveis para todos os controllers que a estendam. Components (sobre os quais você irá aprender mais tarde) são a melhor alternativa para códigos usados por muitos (mas não necessariamente em todos) controllers.

Você pode usar seu `AppController` para carregar components que serão usados em cada controller de sua aplicação. O CakePHP oferece um método `initialize()` que é invocado ao final do construtor do controller para esse tipo de uso:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize()
    {
        // Sempre habilite o CSRF component.
        $this->loadComponent('Csrf');
    }
}
```

Em adição ao método `initialize()`, a antiga propriedade `$components` também vai permitir você declarar quais components devem ser carregados. Enquanto heranças objeto-orientadas normais são enquadradas, os components e helpers usados por um controller são especialmente tratados. Nestes casos, os valores de propriedade do `AppController` são mesclados com arrays de classes controller filhas. Os valores na classe filha irão sempre sobre-escrever aqueles na `AppController`.

Fluxo de requisições

Quando uma requisição é feita para uma aplicação CakePHP, a classe `Cake\Routing\Router` e a classe `Cake\Routing\Dispatcher` usam *Connecting Routes* para encontrar e criar a instância correta do controller. Os dados da requisição são encapsulados em um objeto de requisição. O CakePHP coloca todas as informações importantes de uma requisição na propriedade `$this->request`. Veja a seção *Requisição* para mais informações sobre o objeto de requisição do CakePHP.

Métodos (actions) de controllers

Actions de controllers são responsáveis por converter os parâmetros de requisição em uma resposta para o navegador/usuário que fez a requisição. O CakePHP usa convenções para automatizar este processo e remove alguns códigos clichês que você teria que escrever de qualquer forma.

Por convenção, o CakePHP renderiza uma view com uma versão flexionada do nome da action. Retornando ao nosso exemplo da padaria online, nosso `RecipesController` poderia abrigar as actions `view()`, `share()` e `search()`. O controller seria encontrado em `src/Controller/RecipesController.php` contendo:

```
// src/Controller/RecipesController.php

class RecipesController extends AppController
{
    function view($id)
    {
```

```

        // A lógica da action vai aqui.
    }

    function share($customerId, $recipeId)
    {
        // A lógica da action vai aqui.
    }

    function search($query)
    {
        // A lógica da action vai aqui.
    }
}

```

Os arquivos de template para estas actions seriam **src/Template/Recipes/view.ctp**, **src/Template/Recipes/share.ctp** e **src/Template/Recipes/search.ctp**. A nomenclatura convencional para arquivos view é a versão lowercased (minúscula) e underscored (sem sublinhado) do nome da action.

Actions dos controllers geralmente usam `Controller::set()` para criar um contexto que a View usa para renderizar a camada view. Devido às convenções que o CakePHP usa, você não precisa criar e renderizar as views manualmente. Ao invés, uma vez que uma action de controller é completada, o CakePHP irá manipular a renderização e devolver a view.

Se por alguma razão você quiser pular o comportamento padrão, você pode retornar um objeto `Cake\Network\Response` a partir da action com a resposta definida.

Para que você possa utilizar um controller de forma eficiente em sua própria aplicação, nós iremos cobrir alguns dos atributos e métodos oferecidos pelo controller do core do CakePHP.

Interagindo com views

Os controllers interagem com as views de diversas maneiras. Primeiro eles são capazes de passar dados para as views usando `Controller::set()`. Você também pode decidir no seu controller qual arquivo view deve ser renderizado através do controller.

Definindo variáveis para a view

`Cake\Controller\Controller::set(string $var, mixed $value)`

O método `Controller::set()` é a principal maneira de enviar dados do seu controller para a sua view. Após ter usado o método `Controller::set()`, a variável pode ser acessada em sua view:

```

// Primeiro você passa os dados do controller:

$this->set('color', 'pink');

// Então, na view, você pode utilizar os dados:
?>

Você selecionou a cobertura <?php echo $color; ?> para o bolo.

```

O método `Controller::set()` também aceita um array associativo como primeiro parâmetro. Isto pode oferecer uma forma rápida para atribuir uma série de informações para a view:

```
$data = [
    'color' => 'pink',
    'type' => 'sugar',
    'base_price' => 23.95
];

// Faça $color, $type, e $base_price
// disponíveis na view:

$this->set($data);
```

Renderizando uma view

Cake\Controller\Controller::render(*string \$view, string \$layout*)

O método Controller::render() é chamado automaticamente no fim de cada ação requisitada de um controller. Este método executa toda a lógica da view (usando os dados que você passou usando o método Controller::set()), coloca a view em View::\$layout, e serve de volta para o usuário final.

O arquivo view usado pelo método Controller::render() é determinado por convenção. Se a action search() do controller RecipesController é requisitada, o arquivo view encontrado em **src/Template/Recipes/search.ctp** será renderizado:

```
namespace App\Controller;

class RecipesController extends AppController
{
    // ...
    public function search()
    {
        // Render the view in src/Template/Recipes/search.ctp
        $this->render();
    }
    // ...
}
```

Embora o CakePHP irá chamar o método Controller::render() automaticamente (ao menos que você altere o atributo \$this->autoRender para false) após cada action, você pode usá-lo para especificar um arquivo view alternativo especificando o nome do arquivo view como primeiro parâmetro do método Controller::render().

Se o parâmetro \$view começar com '/', é assumido ser um arquivo view ou elemento relativo ao diretório /src/Template. Isto permite a renderização direta de elementos, muito útil em chamadas AJAX:

```
// Renderiza o elemento em src/Template/Element/ajaxreturn.ctp
$this->render('/Element/ajaxreturn');
```

O segundo parâmetro \$layout do Controller::render() permite que você especifique o layout pelo qual a view é renderizada.

Renderizando uma view específica

Em seu controller você pode querer renderizar uma view diferente do que a convencional. Você pode fazer isso chamando o método Controller::render() diretamente. Uma vez chamado o método Controller::render(), o CakePHP não tentará renderizar novamente a view:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function my_action()
    {
        $this->render('custom_file');
    }
}
```

Isto renderizaria o arquivo `src/Template/Posts/custom_file.ctp` ao invés de `src/Template/Posts/my_action.ctp`

Você também pode renderizar views de plugins utilizando a seguinte sintaxe: `$this->render('PluginName.PluginController/custom_file')`. Por exemplo:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function my_action()
    {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

Isto renderizaria `plugins/Users/src/Template/UserDetails/custom_file.ctp`

Redirecionando para outras páginas

`Cake\Controller\Controller::redirect (string|array $url, integer $status)`

O método de controle de fluxo que você vai usar na majoritariamente é `Controller::redirect()`. Este método recebe seu primeiro parâmetro na forma de uma URL relativa do CakePHP. Quando um usuário executar um pedido com êxito, você pode querer redirecioná-lo para uma tela de recepção.

```
public function place_order()
{
    // Logic for finalizing order goes here
    if ($success) {
        return $this->redirect(
            ['controller' => 'Orders', 'action' => 'thanks']
        );
    }
    return $this->redirect(
        ['controller' => 'Orders', 'action' => 'confirm']
    );
}
```

Este método irá retornar a instância da resposta com cabeçalhos apropriados definidos. Você deve retornar a instância da resposta da sua action para prevenir renderização de view e deixar o dispatcher controlar o redirecionamento corrente.

Você também pode usar uma URL relativa ou absoluta como o parâmetro `$url`:

```
return $this->redirect('/orders/thanks');
return $this->redirect('http://www.example.com');
```

Você também pode passar dados para a action:

```
return $this->redirect(['action' => 'edit', $id]);
```

O segundo parâmetro passado no `Controller::redirect()` permite a você definir um código de status HTTP para acompanhar o redirecionamento. Você pode querer usar o código 301 (movido permanentemente) ou 303 (veja outro), dependendo da natureza do redirecionamento.

Se você precisa redirecionar o usuário de volta para a página que fez a requisição, você pode usar:

```
$this->redirect($this->referer());
```

Um exemplo usando seqüências de consulta e hash pareceria com:

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

A URL gerada seria:

```
http://www.example.com/orders/confirm?product=pizza&quantity=5#top
```

Redirecionando para outra action no mesmo Controller

`Cake\Controller\Controller::setAction($action, $args...)`

Se você precisar redirecionar a atual action para uma diferente no *mesmo* controller, você pode usar `Controller::setAction()` para atualizar o objeto da requisição, modificar o template da view que será renderizado e redirecionar a execução para a action especificada:

```
// De uma action delete, você pode renderizar uma página
// de índice atualizada.
$this->setAction('index');
```

Carregando models adicionais

`Cake\Controller\Controller::loadModel(string $modelClass, string $type)`

O método `loadModel` vem a calhar quando você precisa usar um model que não é padrão do controller ou o seu model não está associado com este.:

```
// Em um método do controller.
$this->loadModel('Articles');
$recentArticles = $this->Articles->find('all', [
    'limit' => 5,
    'order' => 'Articles.created DESC'
]);
```


Se você está usando um provedor de tabelas que não os da ORM nativa você pode ligar este sistema de tabelas aos controllers do CakePHP conectando seus métodos de factory:

```
// Em um método do controller.
$this->modelFactory(
    'ElasticIndex',
    ['ElasticIndexes', 'factory']
);
```

Depois de registrar uma tabela factory, você pode usar o `loadModel` para carregar instâncias:

```
// Em um método do controller
$this->loadModel('Locations', 'ElasticIndex');
```

Nota: O `TableRegistry` da ORM nativa é conectado por padrão como o provedor de ‘Tabelas’.

Paginando um model

`Cake\Controller\Controller::paginate()`

Este método é usado para fazer a paginação dos resultados retornados por seus models. Você pode especificar o tamanho da página (quantos resultados serão retornados), as condições de busca e outros parâmetros. Veja a seção [pagination](#) para mais detalhes sobre como usar o método `paginate()`

O atributo `paginate` lhe oferece uma forma fácil de customizar como `paginate()` se comporta:

```
class ArticlesController extends AppController
{
    public $paginate = [
        'Articles' => [
            'conditions' => ['published' => 1]
        ]
    ];
}
```

Configurando components para carregar

`Cake\Controller\Controller::loadComponent($name, $config = [])`

Em seu método `initialize()` do controller você pode definir qualquer component que quiser carregado, e qualquer configuração de dados para eles:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Csrf');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

property `Cake\Controller\Controller::$components`

A propriedade `$components` em seus controllers permitem a você configurar components. Components configurados e suas dependências serão criados pelo CakePHP para você. Leia a seção [Configuring Components](#) para mais

informações. Como mencionado anteriormente, a propriedade `$components` será mesclada com a propriedade definida em cada classe parente do seu controller.

Configurando helpers para carregar

property `Cake\Controller\Controller::$helpers`

Vamos observar como dizer ao controller do CakePHP que você planeja usar classes MVC adicionais:

```
class RecipesController extends AppController
{
    public $helpers = ['Form'];
}
```

Cada uma dessas variáveis são mescladas com seus valores herdados, portanto não é necessário (por exemplo) redeclarar o `FormHelper`, ou qualquer coisa declarada em seu `AppController`.

Ciclo de vida de callbacks em uma requisição

Os controllers do CakePHP vêm equipados com callbacks que você pode usar para inserir lógicas em torno do ciclo de vida de uma requisição:

Cake\Controller\Controller::beforeFilter (*Event \$event*)

Este método é executado antes de cada ação dos controllers. É um ótimo lugar para verificar se há uma sessão ativa ou inspecionar as permissões de um usuário.

Nota: O método `beforeFilter()` será chamado para ações perdidas.

Cake\Controller\Controller::beforeRender (*Event \$event*)

Chamada após a lógica da action de um controller, mas antes da view ser renderizada. Esse callback não é usado frequentemente, mas pode ser necessário se você estiver chamando `Controller\Controller::render()` manualmente antes do final de uma determinada action.

Cake\Controller\Controller::afterFilter ()

Chamada após cada ação dos controllers, e após a completa renderização da view. Este é o último método executado do controller.

Em adição ao ciclo de vida dos callbacks do controller, *Components (Componentes)* também oferece um conjunto de callbacks similares.

Lembre de chamar os callbacks do `AppController` em conjunto com os callbacks dos controllers para melhores resultados:

```
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
}
```

Mais sobre controllers

O Pages Controller

CakePHP é distribuído com o controller **PagesController.php**. Esse controller é simples, seu uso é opcional e normalmente direcionado a prover páginas estáticas. A homepage que você vê logo depois de instalar o CakePHP utiliza esse controller e o arquivo da view fica em **src/Template/Pages/home.ctp**. Se você criar o arquivo **src/Template/Pages/about.ctp**, você poderá acessá-lo em **http://example.com/pages/about**. Fique a vontade para alterar esse controller para atender suas necessidades ou mesmo excluí-lo.

Quando você cria sua aplicação pelo Composer, o `PagesController` vai ser criado na pasta **src/Controller/**.

Components (Componentes)

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁸¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Configuring Components

Authentication

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁸² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

CookieComponent

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁸³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁸¹ <https://github.com/cakephp/docs>

⁸² <https://github.com/cakephp/docs>

⁸³ <https://github.com/cakephp/docs>

Cross Site Request Forgery

Ao habilitar o componente CSRF você obtém proteção contra ataques. [CSRF](#)⁸⁴ ou Cross Site Request Forgery (Falsificação de solicitação entre sites) é uma vulnerabilidade comum nas aplicações web. Esta brecha permite que o atacante capture e responda uma requisição, e as vezes envie dados através de uma requisição usando tags de imagem ou recursos em outros domínios.

O `CsrfComponent` trabalha setando um cookie no navegador do usuário. Quando os formulários são criados com o `Cake\View\Helper\FormHelper`, um input hidden é adicionado contendo o token CSRF. Durante o evento `Controller.startup`, se a requisição for POST, PUT, DELETE ou PATCH o componente irá comparar os dados da requisição e o valor do cookie. Se um deles estiver faltando ou os dois valores forem incompatíveis o componente lançará um `Cake\Network\Exception\InvalidCsrfTokenException`.

Nota: Você sempre deve verificar o método HTTP que está sendo usado antes de tomar uma ação. Você deve *verificar o método HTTP* ou usar `Cake\Http\ServerRequest::allowMethod()` para garantir que o método HTTP correto está sendo usado.

Novo na versão 3.1: O tipo da exceção foi mudado de `Cake\Network\Exception\ForbiddenException` para `Cake\Network\Exception\InvalidCsrfTokenException`.

Obsoleto desde a versão 3.5.0: Você deve usar Cross Site Request Forgery (CSRF) Middleware ao invés do `CsrfComponent`.

Usando o CsrfComponent

Simplismente adicionando o `CsrfComponent` ao array de componentes, você pode se beneficiar da proteção contra CSRF fornecida:

```
public function initialize()
{
    parent::initialize();
    $this->loadComponent('Csrf');
}
```

As configurações podem ser passadas ao componente através das configurações de componentes. As configurações disponíveis são:

- `cookieName` O nome do cookie a ser enviado. O padrão é `csrfToken`.
- `expiry` Quanto tempo o token CSRF deve durar. Padrão para a sessão no navegador. Aceita valores `strtotime` a partir da versão 3.1
- `secure` Se o cookie será ou não setado com a flag de Segurança. Isto é, o cookie só será setado em conexão HTTPS e qualquer tentativa sobre uma conexão HTTP normal irá falhar. O padrão é `false`.
- `field` O nome do campo no formulário a ser checado. O padrão é `_csrfToken`. Mudar esta opção exigirá também configurações no `FormHelper`.

Quando habilitado, você pode acessar o CSRF token atual fazendo uma chamada ao objeto:

```
$token = $this->request->getParam('_csrfToken');
```

⁸⁴ http://en.wikipedia.org/wiki/Cross-site_request_forgery

Integração com o FormHelper

O `CsrfComponent` integra perfeitamente com o `FormHelper`. Toda vez que você cria um formulário com o `FormHelper`, ele irá gerar um `input hidden` contendo o CSRF token.

Nota: Quando usar o `CsrfComponent` você sempre deverá iniciar seus formulários com o `FormHelper`. Senão, você precisará criar o `input hidden` manualmente para cada formulário que fizer.

Proteção CSRF e Requisições AJAX

Além dos dados da requisição, os tokens CSRF podem ser enviados através de um cabeçalho especial `X-CSRF-Token`. Usar um cabeçalho geralmente facilita a integração de tokens CSRF com aplicações javascript, ou APIs baseadas em XML/JSON.

Desabilitando o Componente CSRF para Ações Específicas

Embora não recomendado, você pode querer desabilitar o `CsrfComponent` em certas requisições. Você pode fazer isto usando o `event dispatcher` do controller, no método `beforeFilter()`:

```
public function beforeFilter(Event $event)
{
    $this->eventManager()->off($this->Csrf);
}
```

Flash

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁸⁵ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Security

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)⁸⁶ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁸⁵ <https://github.com/cakephp/docs>

⁸⁶ <https://github.com/cakephp/docs>

Pagination

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁸⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Request Handling

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁸⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁸⁷ <https://github.com/cakephp/docs>

⁸⁸ <https://github.com/cakephp/docs>

Views (Visualização)

class Cake\View\View

Views são o **V** no MVC. *Views* são responsáveis por gerar a saída específica requerida para a requisição. Muitas vezes isso é um formulário Html, XML, ou JSON, mas *streaming* de arquivos e criar PDF's que os usuários podem baixar também são responsabilidades da camada View.

O CakePHP vem com algumas Classes View construídas para manipular os cenários de renderização mais comuns:

- Para criar *webservices* XML ou JSON você pode usar a [Views JSON & XML](#).
- Para servir arquivos protegidos, ou arquivos gerados dinamicamente, você pode usar [Enviando Arquivos](#).
- Para criar multiplas views com temas, você pode usar [Temas](#).

A App View

AppView é sua Classe View default da aplicação. AppView estende a propria Cake\View\View, classe incluída no CakePHP e é definida em **src/View/AppView.php** como segue:

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
}
```

Você pode usar sua AppView para carregar *helpers* que serão usados por todas as views renderizadas na sua aplicação. CakePHP provê um método `initialize()` que é invocado no final do construtor da View para este tipo de uso:

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
```

```
public function initialize()
{
    // Sempre habilita o *helper* MyUtils
    $this->loadHelper('MyUtils');
}
}
```

View Templates

A camada *View* do CakePHP é como você pode falar com seus usuários. A maior parte do tempo suas views irão renderizar documentos HTML/XHTML para os browsers, mas você também pode precisar responder uma aplicação remota via JSON, ou ter uma saída de um arquivo csv para o usuário.

Os arquivos de *template* CakePHP tem a extensão padrão **.ctp** (CakePHP Template) e utiliza a [Sintaxe PHP alternativa](#)⁸⁹ para controlar estruturas e saídas. Esses arquivos contem a lógica necessária para preparar os dados recebidos do *controller* para o formato de apresentação que estará pronto para o seu público.

Echos Alternativos

Echo, ou imprime a variável no seu *template*:

```
<?php echo $variable; ?>
```

Usando o suporte a Tag curta:

```
<?= $variable ?>
```

Estruturas de controle alternativas

Estruturas de controle, como `if`, `for`, `foreach`, `switch`, e `while` podem ser escritas em um formato simplificado. Observe que não há chaves. Ao invés disso, a chave de fim do “foreach” é substituída por `endforeach`. Cada uma das estruturas de controle listadas anteriormente tem uma sintaxe de fechamento similar: `endif`, `endfor`, `endforeach`, e `endwhile`. Observe também que ao invés do uso de ponto e vírgula depois da estrutura do `foreach` (Exceto o último), existem dois pontos.

O bloco a seguir é um exemplo do uso de `foreach`:

```
<ul>
<?php foreach ($todo as $item): ?>
    <li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

Outro exemplo, usando `if/elseif/else`. Note os dois pontos:

```
<?php if ($username === 'sally'): ?>
    <h3>Olá Sally</h3>
<?php elseif ($username === 'joe'): ?>
    <h3>Olá Joe</h3>
```

⁸⁹ <http://php.net/manual/en/control-structures.alternative-syntax.php>


```
<?php else: ?>
<h3>Olá usuário desconhecido</h3>
<?php endif; ?>
```

Se você preferir usar uma linguagem de template como [Twig](#)⁹⁰, uma subclasse da *View* irá ligar sua linguagem de template e o CakePHP.

Arquivos de template são armazenados em **src/Template/**, em uma pasta nomeada com o nome do *controller*, e com o nome da ação a que corresponde. Por exemplo, o arquivo da *View* da ação “view()” do controller *Products*, seria normalmente encontrada em **src/Template/Products/view.ctp**.

A camada *view* do CakePHP pode ser constituída por um número diferente de partes. Cada parte tem diferentes usos, e serão abordadas nesse capítulo:

- **views:** Templates são a parte da página que é única para a ação sendo executada. Eles formam o cerne da resposta da aplicação.
- **elements:** pequenos bits reúsáveis do código da *view*. *Elements* são usualmente renderizados dentro das *views*.
- **layouts:** Arquivos de modelo que contêm código de apresentação que envolve interfaces da sua aplicação. A maioria das *Views* são renderizadas em um layout.
- **helpers:** Essas classes encapsulam lógica de *View* que é necessária em vários lugares na camada *view*. Entre outras coisas, *helpers* em CakePHP podem ajudar você a construir formulários, construir funcionalidades AJAX, paginar dados do *Model*, ou servir *feed* RSS.
- **cells:** Essas classes proveem uma miniatura de funcionalidades de um controller para criar componentes de interface de usuário independentes. Veja a documentação [View Cells \(Células de Visualização\)](#) para mais informações.

Variáveis da View

Quaisquer variáveis que você definir no seu controller com `set()` ficarão disponíveis tanto na *view* quanto no layout que sua *view* renderiza. Além do mais, quaisquer variáveis definidas também ficarão disponíveis em qualquer *element*. Se você precisa passar variáveis adicionais da *view* para o layout você pode chamar o `set()` no template da *view*, ou use os [Usando View Blocks](#).

Você deve lembrar de **sempre** escapar dados do usuário antes de fazer a saída, pois, o CakePHP não escapa automaticamente a saída. Você pode escapar o conteúdo do usuário com a função `h()`:

```
<?= h($user->bio); ?>
```

Definindo Variáveis da View

`Cake\View\View::set(string $var, mixed $value)`

Views tem um método `set()` que é análogo com o `set()` encontrado nos objetos *Controller*. Usando `set()` no arquivo da sua *view* irá adicionar as variáveis para o layout e *elements* que irão ser renderizadas mais tarde. Veja [Definindo variáveis para a view](#) para mais informações para usar o método `set()`.

Em seu arquivo da *view* você pode fazer:

```
$this->set('activeMenuButton', 'posts');
```

Então, em seu layout, a variável `$activeMenuButton` ficará disponível e conterá o valor ‘posts’.

⁹⁰ <http://twig.sensiolabs.org>

Estendendo Views

Estender Views permite a você utilizar uma view em outra. Combinando isso com os *view blocks* dá a você uma forma poderosa para deixar suas views *DRY*. Por Exemplo, sua aplicação tem uma *sidebar* que precisa mudar dependendo da view específica que está sendo renderizada. Ao estender um arquivo de exibição comum, Você pode evitar repetir a marcação comum para sua barra lateral e apenas definir as partes que mudam:

```
<!-- src/Template/Common/view.ctp -->
<h1><?= $this->fetch('title') ?></h1>
<?= $this->fetch('content') ?>

<div class="actions">
    <h3>Related actions</h3>
    <ul>
        <?= $this->fetch('sidebar') ?>
    </ul>
</div>
```

O arquivo *view* acima poderia ser usado como uma view pai. Espera-se que a view estendida irá definir os *blocks sidebar* e *title*. O *block content* é um *block* especial que o CakePHP cria. Irá conter todo o conteúdo não capturado da view estendida. Assumindo que nosso arquivo da *view* tem a variável `$post` com os dados sobre nosso *post*, a view poderia se parecer com isso:

```
<!-- src/Template/Posts/view.ctp -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post);

$this->start('sidebar');
?>
<li>
    <?php
    echo $this->Html->link('edit', [
        'action' => 'edit',
        $post->id
    ]); ?>
</li>
<?php $this->end(); ?>

// O conteúdo restante estará disponível como o bloco 'content'
// na view pai.
<?= h($post->body) ?>
```

A *view* do post acima mostra como você pode estender uma view, e preencher um conjunto de *blocks*. Qualquer elemento que ainda não esteja em um *block* definido será capturado e colocado em um *block* especial chamado *content*. Quando uma *view* contém uma chamada `extend()`, a execução continua até o final do arquivo da *view* atual. Uma vez completado, a view estendida será renderizada. Chamando `extend()` mais de uma vez em um arquivo da *view* irá substituir a view pai processada em seguida:

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

A view acima irá resultar em `/Common/index.ctp` sendo renderizada como a *view* para a view atual.

Você pode aninhar as view estendidas quantas vezes achar necessário. Cada *view* pode estender outra *view* se necessário. Cada *view* pai irá pegar o conteúdo da *view* anterior com o *block content*.

Nota: Você deve evitar usar `content` como um nome de bloco em seu aplicativo. O CakePHP usa isso para conteúdo não capturado em exibições estendidas.

Você pode resgatar a lista de todos os blocos populados usando o método `blocks()`:

```
$list = $this->blocks();
```

Usando View Blocks

View blocks provê uma API flexível que lhe permite definir slots ou blocos em suas *views/layouts* que serão definidas em outro lugar. Por exemplo, *blocks* são ideais para implementar coisas como *sidebars*, ou regiões para carregar *assets* ao final/início do seu *layout*. *Blocks* podem ser definidos de duas formas: Capturando um bloco, ou por atribuição direta. Os métodos `start()`, `append()`, `prepend()`, `assign()`, `fetch()`, e `end()` permitem que você trabalhe capturando blocos:

```
// Cria o bloco *sidebar*.
$this->start('sidebar');
echo $this->element('sidebar/recent_topics');
echo $this->element('sidebar/recent_comments');
$this->end();

// Anexa ao *sidebar* posteriormente.
$this->start('sidebar');
echo $this->fetch('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();
```

Você também pode anexar em um *block* usando `append()`:

```
$this->append('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();

// O mesmo que acima.
$this->append('sidebar', $this->element('sidebar/popular_topics'));
```

Se você precisa limpar ou sobrescrever um *block* há algumas alternativas. O método `reset()` irá limpar ou sobrescrever um bloco em qualquer momento. O método `assign()` com uma string vazia também pode ser usado para limpar um *block* específico.:

```
// Limpa o conteúdo anterior do *block* sidebar.
$this->reset('sidebar');

// Atribuir uma string vazia também limpará o bloco *sidebar*.
$this->assign('sidebar', '');
```

Novo na versão 3.2: `View::reset()` foi adicionado na versão 3.2

Atribuir um conteúdo de um *block* muitas vezes é usado quando você quer converter uma variável da *view* em um bloco. Por exemplo, você pode querer usar um *block* para a página Título e às vezes atribuir o título como uma variável da *view* no *controller*:

```
// No arquivo da *view* ou *layout* acima $this->fetch('title')
$this->assign('title', $title);
```

O método `prepend()` permite que você prefixe conteúdo para um *block* existente:

```
// Prefixa para *sidebar*
$this->prepend('sidebar', 'this content goes on top of sidebar');
```

Nota: Você deve evitar usar `content` como um nome de *block*. Isto é utilizado pelo CakePHP Internamente para exibições estendidas e exibir conteúdo no layout.

Exibindo *Blocks*

Você pode exibir *blocks* usando o método `fetch()`. `fetch()` irá dar saída ao *block*, retornando '' se um *block* não existir:

```
<?= $this->fetch('sidebar') ?>
```

Você também pode usar *fetch* para condicionalmente mostrar o conteúdo que deverá caso o *block* existir. Isso é útil em layouts, ou estender view onde você quer condicionalmente mostrar títulos ou outras marcações:

```
// In src/Template/Layout/default.ctp
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Menu options</h3>
    <?= $this->fetch('menu') ?>
</div>
<?php endif; ?>
```

Você também pode fornecer um valor padrão para o bloco se ele não existir. Isso lhe permiti adicionar um conteúdo *placeholder* quando o *block* não existe. Você pode definir um valor default usando o segundo argumento:

```
<div class="shopping-cart">
    <h3>Your Cart</h3>
    <?= $this->fetch('cart', 'Seu carrinho está vazio') ?>
</div>
```

Usando *Blocks* para arquivos de script e css

O `HtmlHelper` se baseia em *view blocks*, e os métodos `script()`, `css()`, e `meta()` cada um atualizam um bloco com o mesmo nome quando usados com a opção `block = true`:

```
<?php
// No seu arquivo da *view*
$this->Html->script('carousel', ['block' => true]);
$this->Html->css('carousel', ['block' => true]);
?>

// No seu arquivo do *layout*.
<!DOCTYPE html>
<html lang="en">
    <head>
        <title><?= $this->fetch('title') ?></title>
        <?= $this->fetch('script') ?>
        <?= $this->fetch('css') ?>
```

```
</head>
// Restante do seu layout abaixo
```

O `Cake\View\Helper\HtmlHelper` também lhe permite controlar qual *block* o css ou script irá:

```
// Na sua *view*
$this->Html->script('carousel', ['block' => 'scriptBottom']);

// No seu *layout*
<?= $this->fetch('scriptBottom') ?>
```

Layouts

Um layout contém códigos de apresentação que envolvem uma *view*. Qualquer coisa que você quer ver em todas as suas *views* deve ser colocada em um *layout*.

O layout default do CakePHP está localizado em **src/Template/Layout/default.ctp**. Se você quer alterar a aparência geral da sua aplicação, então este é o lugar certo para começar, porque o código de exibição processado pelo controlador é colocado dentro do layout padrão quando a página é processada.

Outros arquivos de *layout* devem estar localizados em **src/Template/Layout**. Quando você cria um *layout*, você precisa dizer para o cakePHP onde colocar o resultado de suas *views*. Para fazer isso, tenha certeza que seu *layout* inclui um lugar para `$this->fetch('content')`. Aqui um exemplo do que um layout padrão pode parecer:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title><?= h($this->fetch('title')) ?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Inclui arquivos externos e scripts aqui. (Veja HTML helper para mais informações.
  ↪) -->
<?php
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
?>
</head>
<body>

<!-- Se você quiser algum tipo de menu para mostrar no topo
de todas as suas *views*, inclua isso aqui -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Aqui é onde eu quero que minhas views sejam exibidas -->
<?= $this->fetch('content') ?>

<!-- Adicione um rodapé para cada página exibida -->
<div id="footer">...</div>

</body>
</html>
```

Os blocos `script`, `css` e `meta` contém qualquer conteúdo definido nas *views* usando o HTML helper do CakePHP. Útil para incluir arquivos JavaScript e CSS das suas *views*.

Nota: Quando usado `HtmlHelper::css()` ou `HtmlHelper::script()` em arquivos de template, especifique `'block' => true` para colocar o código HTML em um bloco com o mesmo nome. (Veja API para mais detalhes de como utilizar).

O bloco `content` contém os conteúdos da view renderizada.

Você pode definir o conteúdo do bloco `title` de dentro do seu arquivo da *view*:

```
$this->assign('title', 'Visualizar Usuários Ativos');
```

Você pode criar quantos layouts você quiser: somente os coloque no diretório **src/Template/Layout**, a troca entre eles dentro das suas ações do *controller* ocorre usando a propriedade do *controller* ou *view* `$layout`:

```
// Em um controller
public function admin_view()
{
    // Define o layout.
    $this->viewBuilder()->setLayout('admin');

    // Antes da versão 3.4
    $this->viewBuilder()->layout('admin');

    // Antes da versão 3.1
    $this->layout = 'admin';
}

// Em um arquivo de *view*
$this->layout = 'loggedin';
```

Por exemplo, se uma seção de meu site inclui um pequeno espaço para um banner de propaganda, Eu devo criar um novo layout com o pequeno espaço de propaganda e especificá-lo para todas as ações dos *controllers* usando algo parecido com:

```
namespace App\Controller;

class UsersController extends AppController
{
    public function viewActive()
    {
        $this->set('title', 'Visualizar Usuários Ativos');
        $this->viewBuilder()->setLayout('default_small_ad');

        // Antes da versão 3.4
        $this->viewBuilder()->layout('default_small_ad');

        // Antes da versão 3.1
        $this->layout = 'default_small_ad';
    }

    public function viewImage()
    {
        $this->viewBuilder()->setLayout('image');

        // Exibe a imagem do usuário
    }
}
```

Além do layout padrão, A aplicação esqueleto CakePHP também tem um layout 'ajax'. O layout Ajax é útil para criar resposta AJAX - É um layout vazio. (A maioria das chamadas AJAX somente necessitam retornar uma porção de marcação, ao invés de uma interface totalmente renderizada.)

A aplicação esqueleto também tem um layout padrão para ajudar a gerar RSS.

Usando Layouts de Plugins

Se você quer usar um layout existente em um plugin, você pode usar *sintaxe plugin*. Por exemplo, para usar o layout contact do plugin Contacts:

```
namespace App\Controller;

class UsersController extends AppController
{
    public function view_active()
    {
        $this->viewBuilder()->layout('Contacts.contact');
        // ou o seguinte para a versão anterior a 3.1
        $this->layout = 'Contacts.contact';
    }
}
```

Elements

Cake\View\View::element (string \$elementPath, array \$data, array \$options = [])

Muitas aplicações tem pequenos blocos de código de apresentação que precisam ser repetidos página a página, algumas vezes em diferentes lugares do layout. O CakePHP pode ajudar você repetir partes do seu website que precisam ser reusadas. Essas partes reusáveis são chamadas de Elements. Publicidade, Caixas de ajuda, controles de navegação, menus extras, formulários de login, e callouts são muitas vezes implementados em CakePHP como *elements*. Um elemento é basicamente uma mini-view que pode ser incluída em outras *views*, em *layouts*, e mesmo em outros *elements*. *Elements* podem ser usados para fazer uma *view* mais legível, colocando a renderização de elementos repetitivos em seu próprio arquivo. Eles também podem ajudá-lo a reusar conteúdos fragmentados em sua aplicação.

Elements estão na pasta **src/Template/Element/**, e tem a extensão .ctp. Eles são exibidos usando o método *element* da *view*:

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

Passando variáveis para um Element

Você pode passar dados para um *element* através do segundo parâmetro do método *element*:

```
echo $this->element('helpbox', [
    "helptext" => "Ah, Este texto é muito útil."
]);
```

Dentro do arquivo do *element*, todas as variáveis estarão disponíveis como membros de um array de parâmetros (da mesma forma que Controller::set() no *controller* funciona com arquivos de template). No exemplo a seguir, no arquivo **src/Template/Element/helpbox.ctp** pode usar a variável \$helptext:

```
// Dentro do arquivo src/Template/Element/helpbox.ctp
echo $helptext; // Resulta em "Ah, Esse texto muito útil."
```

O método `View::element()` também suporta opções para o elemento. As opções suportadas são ‘cache’ e ‘callbacks’. Um exemplo:

```
echo $this->element('helpbox', [
    "helptext" => "Isso é passado para o *element* como $helptext",
    "foobar" => "Isso é passado para o *element* como $foobar",
],
[
    // Usa a configuração de cache "long_view"
    "cache" => "long_view",
    // Define como true para ter before / afterRender chamado para o elemento
    "callbacks" => true
]);
```

Cache de Elementos é facilitado através da Classe Cache. Você pode configurar *elements* para serem armazenados em qualquer configuração de cache que você possua. Isso dá a você uma grande flexibilidade para decidir onde e por quanto tempo *elements* serão armazenados. Para fazer cache de diferentes versões do mesmo *element* em uma aplicação, forneça um valor para a chave única de cache usando o seguinte formato:

```
$this->element('helpbox', [], [
    "cache" => ['config' => 'short', 'key' => 'unique value']
]);
```

Se você precisa de mais lógica em seu *element*, como dados dinâmicos de uma fonte de dados, considere usar uma *View Cell* ao invés de um *element*. Encontre mais *sobre View Cells*.

Fazendo Cache de *Elements*

Você pode tirar vantagem do cache de *view* do CakePHP se você fornecer um parametro de cache. Se definido como `true`, isso irá fazer cache do *element* na Configuração de cache ‘default’. De qualquer forma, você pode escolher a configuração de cache que será usada. Veja *Caching* para mais informações ao configurar Cache. Um simples exemplo de caching de *element* poderia ser:

```
echo $this->element('helpbox', [], ['cache' => true]);
```

Se você renderizar o mesmo *element* mais de uma vez em uma *view* e tiver o cache habilitado, tenha certeza de definir o parâmetro ‘key’ com um nome diferente a cada vez. Isso impedirá que cada chamada sucessiva sobrescreva o resultado do cache do *element* anterior. Por exemplo:

```
echo $this->element(
    'helpbox',
    ['var' => $var],
    ['cache' => ['key' => 'first_use', 'config' => 'view_long']]
);

echo $this->element(
    'helpbox',
    ['var' => $differenVar],
    ['cache' => ['key' => 'second_use', 'config' => 'view_long']]
);
```


O bloco acima assegurará que o resultado dos *elements* terão o cache armazenados separadamente. Se você quer todos os *elements* usando a mesma configuração de cache, você pode evitar a repetição definindo `View::$elementCache` para a configuração que deseja utilizar. O CakePHP irá usar essa configuração quando nenhuma for fornecida.

Requisitando *Elements* de um plugin

Se você está usando um plugin e deseja usar *elements* de dentro do plugin, simplesmente use a familiar *sintaxe plugin*. Se a *view* está sendo renderizada de um controller/action de um plugin, o nome do plugin será automaticamente prefixado em todos os *elements* a não ser que outro nome de plugin esteja presente. Se o *element* não existe no plugin, irá buscar na pasta principal da aplicação:

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

Se sua *view* é uma parte de um plugin, você pode omitir o nome do plugin. Por exemplo, se você está em `ContactsController` do plugin `Contacts`, terá o seguinte:

```
echo $this->element('helpbox');
// and
echo $this->element('Contacts.helpbox');
```

São equivalentes e irá resultar no mesmo elementos sendo renderizado.

Para *elements* dentro de uma subpasta de um plugin (e.g., `plugins/Contacts/Template/Element/sidebar/helpbox.ctp`), use o seguinte:

```
echo $this->element('Contacts.sidebar/helpbox');
```

Requisitando *Elements* do App

Se você está dentro de um arquivos de template de um plugin e quer renderizar um *element* residido em sua aplicação principal ou outro plugin, use o seguinte:

```
echo $this->element('some_global_element', [], ['plugin' => false]);
// or...
echo $this->element('some_global_element', ['localVar' => $someData], ['plugin' => false]);
```

Routing prefix e *Elements*

Novo na versão 3.0.1.

Se você tiver um Routing prefix configurado, o caminho do *Element* pode ser trocado para a localização do prefixo, como layouts e *actions* da *View* fazem. Assumindo que você tem um prefixo “Admin” configurado e você chama:

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

O primeiro *element* procurado será em `src/Template/Admin/Element/`. Se o arquivo não existir, será procurado na localização padrão.

Fazendo Cache de Seções da sua View

`Cake\View\View::cache` (*callable \$block, array \$options = []*)

As vezes gerar uma seção do resultado da sua view pode ser custoso porque foram renderizados *View Cells (Células de Visualização)* ou operações de *helper's* custosas. Para ajudar sua aplicação a rodar mais rapidamente o CakePHP fornece uma forma de fazer cache de seções de view:

```
// Assumindo algumas variáveis locais
echo $this->cache(function () use ($user, $article) {
    echo $this->cell('UserProfile', [$user]);
    echo $this->cell('ArticleFull', [$article]);
}, ['key' => 'my_view_key']);
```

Por padrão um conteúdo da view em cache irá ir para a configuração de cache `View::$elementCache`, mas você pode usar a opção `config` para alterar isso.

Eventos da View

Como no *Controller*, *view* dispara vários eventos/callbacks que você pode usar para inserir lógica em torno do ciclo de vida da renderização:

Lista de Eventos

- `View.beforeRender`
- `View.beforeRenderFile`
- `View.afterRenderFile`
- `View.afterRender`
- `View.beforeLayout`
- `View.afterLayout`

Você pode anexar à aplicação *event listeners* para esses eventos ou usar *Helper Callbacks*.

Criando suas próprias Classes View

Talvez você precise criar classes *view* personalizadas para habilitar novos tipos de visualizações de dados ou adicione uma lógica de exibição de visualização personalizada adicional à sua aplicação. Como a maioria dos Componentes do CakePHP, as classes view têm algumas convenções:

- Arquivos das Classes View devem ser colocados em **src/View**. Por exemplo: **src/View/PdfView.php**
- Classes View devem ter o sufixo `View`. Por Exemplo: `PdfView`.
- Quando referenciar nome de Classes *view* você deve omitir o sufixo `View`. Por Exemplo:
`$this->viewBuilder()->className('Pdf');`

Você também vai querer estender a Classe `View` para garantir que as coisas funcionem corretamente:

```
// Em src/View/PdfView.php
namespace App\View;

use Cake\View\View;

class PdfView extends View
{
    public function render($view = null, $layout = null)
    {
        // Custom logic here.
    }
}
```

Substituir o método de renderização permite que você tome controle total sobre como seu conteúdo é Processado.

Mais sobre Views

View Cells (Células de Visualização)

View cells são pequenos *mini-controllers* que podem invocar lógica de visualização e renderizar templates. A ideia de *cells* é emprestada das *cells do Ruby*⁹¹, onde desempenham papel e finalidade semelhantes.

Quando usar Cells

Cells são ideais para construir componentes de páginas reutilizáveis que requerem interação com modelos, lógica de visualização, e lógica de renderizaço. Um exemplo simples seria o carinho em uma loja online, ou um menu de navegação *data-driven* em um CMS.

Para criar uma *cell*, defina uma classe em **src/View/Cell** e um *template* em **src/Template/Cell/**. Nesse exemplo, nós estaremos fazendo uma *cell* para exibir o número de mensagens em uma caixa de notificações do usuário. Primeiro, crie o arquivo da classe. O seu conteúdo deve se parecer com:

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{
    public function display()
    {
    }
}
```

Salve esse arquivo dentro de **src/View/Cell/InboxCell.php**. Como você pode ver, como em outras classes no CakePHP, *Cells* tem algumas convenções:

- As *Cells* ficam no *namespace* `App\View\Cell`. Se você está fazendo uma *cell* em um *plugin*, o *namespace* seria `NomeDoPlugin\View\Cell`.
- Os nomes das classes devem terminar com *Cell*.

⁹¹ <https://github.com/apotonick/cells>

- Classes devem herdar de `Cake\View\Cell`.

Nós adicionamos um método `display()` vazio para nossa *cell*; esse é o método padrão convencional quando a *cell* é renderizada. Nós vamos abordar o uso de outros métodos mais tarde na documentação. Agora, crie o arquivo `src/Template/Cell/Inbox/display.ctp`. Esse será nosso *template* para a nossa nova *cell*.

Você pode gerar este esboço de código rapidamente usando o `bake`:

```
bin/cake bake cell Inbox
```

Gera o código que digitamos acima.

Implementando a *Cell*

Assumindo que nós estamos trabalhando em uma aplicação que permite que usuários enviem mensagens um para o outro. Nós temos o modelo `Messages`, e nós queremos mostrar a quantidade de mensagens não lidas sem ter que poluir o *AppController*. Este é um perfeito caso de uso para uma *cell*. Na classe que acabamos de fazer, adicione o seguinte:

```
namespace App\View\Cell;

use Cake\View\Cell;

class InboxCell extends Cell
{
    public function display()
    {
        $this->loadModel('Messages');
        $unread = $this->Messages->find('unread');
        $this->set('unread_count', $unread->count());
    }
}
```

Porque as *Cells* usam o `ModelAwareTrait` e o `ViewVarsTrait`, Elas tem um comportamento muito parecido com um *controller*. Nós podemos usar os métodos `loadModel()` e `set()` como faríamos em um *controller*. Em nosso arquivo de *template*, adicione o seguinte:

```
<!-- src/Template/Cell/Inbox/display.ctp -->
<div class="notification-icon">
    Você tem <?= $unread_count ?> mensagem não lidas.
</div>
```

Nota: *Cell templates* têm um escopo isolado que não compartilha a mesma instância da view utilizada para processar o *template* e o *layout* para o *controller* ou outras *cells*. Assim, eles não sabem de nenhuma chamada de helper feita ou blocos definidos no *template* / *layout* da *action* e vice-versa.

Carregando *Cells*

Cells podem ser carregadas nas *views* usando o método `cell()` e funciona da mesma forma em ambos os contextos:

```
// Carrega uma *cell* da aplicação
$cell = $this->cell('Inbox');

// Carrega uma *cell* de um plugin
$cell = $this->cell('Messaging.Inbox');
```

O código acima irá carregar a célula nomeada e executar o método `display()`. Você pode executar outros método usando o seguinte:

```
// Executa o método *Run* na *cell* *Inbox*
$cell = $this->cell('Inbox::expanded');
```

Se você precisa de lógica no *controller* para decidir quais *cells* serão carregadas em uma requisição, você pode usar o `CellTrait` no seu *controller* para habilitar o método `cell()` lá:

```
namespace App\Controller;

use App\Controller\AppController;
use Cake\View\CellTrait;

class DashboardsController extends AppController
{
    use CellTrait;

    // More code.
}
```

Passando argumento para a *Cell*

Você muitas vezes vai querer parametrizar métodos da *cell* para fazer *cells* mais flexíveis. Usando o segundo e terceiro argumento do método `cell()`, você pode passar parâmetros de ação e opções adicionais para suas classes de *cell*, como um array indexado:

```
$cell = $this->cell('Inbox::recent', ['-3 days']);
```

O código acima corresponderia a seguinte assinatura de função:

```
public function recent($since)
{
}
```

Renderizando uma *Cell*

Uma vez a célula carregada e executada, você provavelmente vai querer renderizá-la. A maneira mais fácil para renderizar uma *cell* é dando um *echo*:

```
<?= $cell ?>
```

Isso irá renderizar o *template* correspondente a versão minúscula e separada com underscore do nome da nossa action, e.g. **display.ctp**.

Porque as *cells* usam `View` para renderizar *templates*, você pode carregar *cells* adicionais dentro do *template* da *cell* se necessário.

Nota: O *echo* da *cell* usa o método PHP mágico `__toString()` para prevenir o PHP de mostrar o nome do arquivo e o número da linha caso algum erro fatal seja disparado. Para obter uma mensagem de erro significativa, é recomendado usar o método `Cell::render()`, por exemplo `<?= $cell->render() ?>`.

Renderizando template alternativos

Por convenção *cells* renderizam *templates* que correspondem a *action* que está sendo executada. Se você precisar renderizar um *template* de visualização diferente, você pode especificar o *template* para usar quando estiver renderizando a *cell*:

```
// Chamando render() explicitamente
echo $this->cell('Inbox::recent', ['-3 days'])->render('messages');

// Especificando o template antes de executar *echo* da *cell*
$cell = $this->cell('Inbox');
$cell->template = 'messages';
echo $cell;
```

Caching Cell Output

Ao renderizar uma célula, você pode querer armazenar em cache a saída renderizada se o conteúdo não mudar frequentemente ou para ajudar a melhorar o desempenho do sua aplicação. Você pode definir a opção *cache* ao criar uma célula para ativar e configurar o cache:

```
// Faz cache usando a configuração padrão e uma chave gerada
$cell = $this->cell('Inbox', [], ['cache' => true]);

// Faz cache usando uma configuração específica a uma chave gerada
$cell = $this->cell('Inbox', [], ['cache' => ['config' => 'cell_cache']]);

// Especificando a chave e a configuração utilizada
$cell = $this->cell('Inbox', [], [
    'cache' => ['config' => 'cell_cache', 'key' => 'inbox_' . $user->id]
]);
```

Se uma chave é gerada a versão sublinhada da classe da *cell* e o nome do *template* serão usados

Nota: Uma nova instância da *View* é usada para cada *cell* e esses novos objetos não compartilham o contexto com o *template* principal/*layout*. Cada *cell* é *self-contained* e somente tem acesso as variáveis passadas como argumento pelo chamada do método `View::cell()`.

Temas

Temas no CakePHP são simplesmente plugins que focam em prover arquivos de *template*. Veja a seção em [Criando seus próprios complementos](#). Você pode tirar vantagem de temas, deixando fácil a troca da aparência da sua página rapidamente. Além de arquivos de *templates*, eles também podem prover *helpers* e ‘*cells*’ se o seu tema assim requerer. Quando usado *cells* e *helpers* no seu tema, você precisará continuar usando a *sintaxe plugin*.

Para usar temas, defina o tema na *action* do seu *controller* ou no método `beforeRender()`:

```

class ExamplesController extends AppController
{
    // Para o CakePHP antes da versão 3.1
    public $theme = 'Modern';

    public function beforeRender(\Cake\Event\Event $event)
    {
        $this->viewBuilder()->setTheme('Modern');

        // Para o cakePHP antes da versão 3.5
        $this->viewBuilder()->theme('Modern');
    }
}

```

Os arquivos de template do tema precisam estar dentro de um plugin com o mesmo nome. Por exemplo, o tema acima seria encontrado em **plugins/Modern/src/Template**. É importante lembrar que o CakePHP espera o nome do plugin/tema em PascalCase. Além de que, a estrutura da pasta dentro da pasta **plugins/Modern/src/Template** é exatamente o mesmo que **src/Template/**.

Por exemplo, o arquivo de exibição para uma *action* de edição de um controller de posts residiria em **plugins/Modern/src/Template/Posts/edit.ctp**. Os arquivos de layout residiriam em **plugins/Modern/src/Template/Layout/**. Você pode fornecer modelos personalizados para plugins com um tema também. Se você tiver um plugin chamado 'Cms', que contenha um TagsController, o tema moderno poderia fornecer **plugins/Modern/src/Template/Plugin/Cms/Tags/edit.ctp** para substituir o template da edição no plugin.

Se um arquivo de exibição não puder ser encontrado no tema, o CakePHP tentará localizar a visualização arquivo na pasta **src/Template/**. Desta forma, você pode criar arquivos de *template* mestre e simplesmente substituí-los caso a caso na pasta do seu tema.

Assets do Tema

Como os temas são plugins CakePHP padrão, eles podem incluir qualquer asset necessário em seu diretório webroot. Isso permite uma fácil embalagem e distribuição de temas. Enquanto estiver em desenvolvimento, requisições de assets do tema serão manipuladas por: `php:class:Cake\Routing\Dispatcher`. Para melhorar o desempenho para ambientes de produção, é recomendável que você *Aprimorar a performance de sua aplicação*.

Todos os ajudantes internos do CakePHP estão cientes de temas e criará o Corrija os caminhos automaticamente. Como arquivos de template, se um arquivo não estiver na pasta do tema, será padrão para a pasta webroot principal:

```

// Em um tema com o nome 'purple_cupcake'
$this->Html->css('main.css');

// crie os diretórios como
/purple_cupcake/css/main.css

// e crie o link como
plugins/PurpleCupcake/webroot/css/main.css

```

Views JSON & XML

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁹² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Helpers (Facilitadores)

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁹³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Breadcrumbs

```
class Cake\View\Helper\BreadcrumbsHelper (View $view, array $config = [])
```

Novo na versão 3.3.6.

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁹⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Flash

```
class Cake\View\Helper\FlashHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁹⁵ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁹² <https://github.com/cakephp/docs>

⁹³ <https://github.com/cakephp/docs>

⁹⁴ <https://github.com/cakephp/docs>

⁹⁵ <https://github.com/cakephp/docs>

Form

```
class Cake\View\Helper\FormHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)⁹⁶ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Criando Inputs para Dados Associados

Html

```
class Cake\View\Helper\HtmlHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)⁹⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Number

```
class Cake\View\Helper\NumberHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)⁹⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Paginator

```
class Cake\View\Helper\PaginatorHelper (View $view, array $config = [])
```

⁹⁶ <https://github.com/cakephp/docs>

⁹⁷ <https://github.com/cakephp/docs>

⁹⁸ <https://github.com/cakephp/docs>

PaginatorHelper Templates

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)⁹⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

RSS

```
class Cake\View\Helper\RssHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Session

```
class Cake\View\Helper\SessionHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Text

```
class Cake\View\Helper\TextHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

⁹⁹ <https://github.com/cakephp/docs>

¹⁰⁰ <https://github.com/cakephp/docs>

¹⁰¹ <https://github.com/cakephp/docs>

¹⁰² <https://github.com/cakephp/docs>

Time

```
class Cake\View\Helper\TimeHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Url

```
class Cake\View\Helper\UrlHelper (View $view, array $config = [])
```

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Classe *Helper*

¹⁰³ <https://github.com/cakephp/docs>

¹⁰⁴ <https://github.com/cakephp/docs>

Models (Modelos)

Models (Modelos) são as classes que servem como camada de negócio na sua aplicação. Isso significa que eles devem ser responsáveis pela gestão de quase tudo o que acontece em relação a seus dados, sua validade, interações e evolução do fluxo de trabalho de informação no domínio do trabalho.

No CakePHP seu modelo de domínio da aplicação é dividido em 2 tipos de objetos principais. Os primeiros são **repositories (repositórios)** ou **table objects (objetos de tabela)**. Estes objetos fornecem acesso a coleções de dados. Eles permitem a você salvar novos registros, modificar/deletar os que já existem, definir relacionamentos, e executar operações em massa. O segundo tipo de objetos são as **entities (entidades)**. Entities representam registros individuais e permitem a você definir comportamento em nível de linha/registro e funcionalidades.

O ORM (MOR - Mapeamento Objeto-Relacional) nativo do CakePHP especializa-se em banco de dados relacionais, mas pode ser estendido para suportar fontes de dados alternativas.

O ORM do Cakephp toma emprestadas ideias e conceitos dos padrões ActiveRecord e Datamapper. Isso permite criar uma implementação híbrida que combina aspectos de ambos padrões para criar uma ORM rápida e simples de utilizar.

Antes de nós começarmos explorando o ORM, tenha certeza que você *configure your database connections*.

Nota: Se você é familiarizado com versões anteriores do CakePHP, você deveria ler o *Guia de atualização para o novo ORM* para esclarecer diferenças importantes entre o CakePHP 3.0 e suas versões antigas.

Exemplo rápido

Para começar você não precisa escrever código. Se você seguiu as convenções do CakePHP para suas tabelas de banco de dados, você pode simplesmente começar a usar o ORM. Por exemplo, se quiséssemos carregar alguns dados da nossa tabela `articles` poderíamos fazer:

```
use Cake\ORM\TableRegistry;
$articles = TableRegistry::get('Articles');
$query = $articles->find();
foreach ($query as $row) {
    echo $row->title;
}
```

Nota-se que nós não temos que criar qualquer código ou definir qualquer configuração. As convenções do CakePHP nos permitem pular alguns códigos clichê, e permitir que o framework insira classes básicas enquanto sua aplicação

não criou uma classe concreta. Se quiséssemos customizar nossa classe `ArticlesTable` adicionando algumas associações ou definir alguns métodos adicionais, deveríamos acrescentar o seguinte a `src/Model/Table/ArticlesTable.php` após a tag de abertura `<?php`:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
}
```

Classes de tabela usam a versão CamelCased do nome da tabela com o sufixo `Table` como o nome da classe. Uma vez que sua classe fora criada, você recebe uma referência para esta utilizando o `ORM\TableRegistry` como antes:

```
use Cake\ORM\TableRegistry;
// Agora $articles é uma instância de nossa classe ArticlesTable.
$articles = TableRegistry::get('Articles');
```

Agora que temos uma classe de tabela concreta, nós provavelmente vamos querer usar uma classe de entidade concreta. As classes de entidade permitem definir métodos de acesso, métodos mutantes, definir lógica personalizada para os registros individuais e muito mais. Vamos começar adicionando o seguinte para `src/Model/Entity/Article.php` após a tag de abertura `<?php`:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
}
```

Entidades usam a versão singular CamelCase do nome da tabela como seu nome de classe por padrão. Agora que nós criamos nossa classe de entidade, quando carregarmos entidades do nosso banco de dados, nós iremos receber instâncias da nossa nova classe `Article`:

```
use Cake\ORM\TableRegistry;

// Agora uma instância de ArticlesTable.
$articles = TableRegistry::get('Articles');
$query = $articles->find();

foreach ($query as $row) {
    // Cada linha é agora uma instância de nossa classe Article.
    echo $row->title;
}
```

CakePHP utiliza convenções de nomenclatura para ligar as classes de tabela e entidade juntas. Se você precisar customizar qual entidade uma tabela utiliza, você pode usar o método `entityClass()` para definir nomes de classe específicos.

Veja os capítulos em *Objetos de tabela* e *Entidades* para mais informações sobre como usar objetos de tabela e entidades em sua aplicação.

Mais informação

O básico sobre banco de dados

A camada de acesso a banco de dados do CakePHP abstrai e fornece auxílio com a maioria dos aspectos de lidar com bancos de dados relacionais como, manter conexões com o servidor, contruir consultas, prevenir injeções SQL, inspecionar e alterar schemas, e com debugging e profiling de consultas enviadas ao banco de dados.

Tour Rápido

As funções descritas nesse capítulo ilustram o que é possível fazer com a API de acesso a banco de dados de baixo-nível. Se ao invés, você deseja aprender mais sobre o ORM completo, você pode ler as seções *Construtor de queries* e *Objetos de tabela*.

A maneira mais fácil de criar uma conexão de banco de dados é usando uma string DSN:

```
use Cake\Datasource\ConnectionManager;

$dsn = 'mysql://root:password@localhost/my_database';
ConnectionManager::config('default', ['url' => $dsn]);
```

Uma vez criada, você pode acessar o objeto da conexão para iniciar a usá-lo:

```
$connection = ConnectionManager::get('default');
```

Bancos de Dados Suportados

O CakePHP suporta os seguintes servidores de banco de dados relacionais:

- MySQL 5.1+
- SQLite 3
- PostgreSQL 8+
- SQLServer 2008+
- Oracle (atravéz de um plugin da comunidade)

Você precisará da extensão PDO correta instalada para cada um dos drivers de banco de dados acima. As APIs processuais não são suportadas.

O banco de dados Oracle é suportado através do plugin da comunidade [Driver para Banco de Dados Oracle](#)¹⁰⁵.

Executando Instruções de Consulta

Executar consultas SQL é uma moleza:

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
$results = $connection->execute('SELECT * FROM articles')->fetchAll('assoc');
```

¹⁰⁵ <https://github.com/CakeDC/cakephp-oracle-driver>

Você pode usar prepared statement para inserir parâmetros:

```
$results = $connection
->execute('SELECT * FROM articles WHERE id = :id', ['id' => 1])
->fetchAll('assoc');
```

Também é possível usar tipos de dados complexos como argumentos:

```
$results = $connection
->execute(
    'SELECT * FROM articles WHERE created >= :created',
    ['created' => DateTime('1 day ago')],
    ['created' => 'datetime']
)
->fetchAll('assoc');
```

Ao invés de escrever a SQL manualmente, você pode usar o query builder:

```
$results = $connection
->newQuery()
->select('*')
->from('articles')
->where(['created >' => new DateTime('1 day ago'), ['created' => 'datetime']])
->order(['title' => 'DESC'])
->execute()
->fetchAll('assoc');
```

Executando Instruções de Inserção

Inserir registros no banco de dados é geralmente uma questão de algumas linhas:

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
$connection->insert('articles', [
    'title' => 'A New Article',
    'created' => new DateTime('now')
], ['created' => 'datetime']);
```

Executando Instruções de Atualização

Atualizar registros no banco de dados é igualmente intuitivo, o exemplo a seguir atualizará o artigo com **id** 10:

```
use Cake\Datasource\ConnectionManager;

$connection = ConnectionManager::get('default');
$connection->update('articles', ['title' => 'New title'], ['id' => 10]);
```

Executando Instruções de Exclusão

Da mesma forma, o método `delete()` é usado para excluir registros do banco de dados, o exemplo a seguir exclui o artigo com **id** 10:


```
use Cake\Datasource\ConnectionManager;
$connection = ConnectionManager::get('default');
$connection->delete('articles', ['id' => 10]);
```

Configuração

Por convenção, as conexões do banco de dados são configuradas em **config/app.php**. As informações de conexão definidas neste arquivo são alimentadas em `Cake\Datasource\ConnectionManager` criando a configuração de conexão que sua aplicação usará. Exemplos de informações de conexão podem ser encontradas em **config/app.default.php**. Uma configuração seria mais ou menos assim:

```
'Datasources' => [
    'default' => [
        'className' => 'Cake\Database\Connection',
        'driver' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'localhost',
        'username' => 'my_app',
        'password' => 'sekret',
        'database' => 'my_app',
        'encoding' => 'utf8',
        'timezone' => 'UTC',
        'cacheMetadata' => true,
    ]
],
```

O exemplo acima criará a conexão 'default', com os parâmetros fornecidos. Você pode definir quantas conexões quiser no seu arquivo de configuração. Você também pode definir conexões adicionais em tempo de execução usando o método `Cake\Datasource\ConnectionManager::config()`. Um bom exemplo disso seria:

```
use Cake\Datasource\ConnectionManager;

ConnectionManager::config('default', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'sekret',
    'database' => 'my_app',
    'encoding' => 'utf8',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
]);
```

As opções de configuração também podem ser fornecidas como uma string *DSN*. Isso é útil ao trabalhar com variáveis de ambiente ou *PaaS* providers:

```
ConnectionManager::config('default', [
    'url' => 'mysql://my_app:sekret@localhost/my_app?encoding=utf8&timezone=UTC&
    ↪cacheMetadata=true',
]);
```

Ao usar uma string DSN, você pode definir qualquer parâmetros/opções adicionais como argumentos de query string.

Por padrão, todos objetos Table usará a conexão `default`. Para usar uma conexão não-padrão, consulte [Configurando Conexões](#).

Existem várias keys suportadas na configuração de banco de dados. Uma lista completa é a seguinte:

className O nome completo de classe incluindo namespace da classe que representa a conexão a um servidor de banco de dados. Esta classe é responsável por carregar o driver do banco de dados, fornecendo mecanismos de transação SQL e preparando instruções SQL entre outras coisas.

driver O nome da classe do driver usado para implementar todas as especificidades para um mecanismo de banco de dados. Isso pode ser um nome de classe curto usando [sintaxe plugin](#), um nome de classe com seu namespace ou uma instância de driver. Exemplos de nomes de classes curtos são `MySQL`, `SQLite`, `Postgres` e `Sqlserver`.

persistent Se deve ou não usar uma conexão persistente com o banco de dados. Esta opção não é suportada pelo `SqlServer`. A partir da versão 3.4.13 do CakePHP, uma exceção é lançada se você tentar definir `persistent` como `true` com `SqlServer`.

host O nome de host do servidor de banco de dados (ou o endereço IP).

username O nome de usuário da conta.

password A senha da conta.

database O nome do banco de dados para essa conexão usar. Evite usar `.` no nome do seu banco de dados. Por causa de como isso complica citação de identificadores, o CakePHP não suporta `.` em nomes de banco de dados. O caminho para o seu banco de dados `SQLite` deve ser um caminho absoluto (ex: `ROOT . DS . 'my_app.db'`) para evitar caminhos incorretos causados por caminhos relativos.

port (opcional) A porta TCP ou o soquete Unix usado para se conectar ao servidor.

encoding Indica a configuração de charset usado ao enviar instruções SQL ao servidor. Seu padrão é a codificação padrão do banco de dados para todos os banco de dados exceto o `DB2`. Se você deseja usar a codificação `UTF-8` com conexões `MySQL`, você deve usar `'utf8'` sem o hífen.

timezone Fuso horário do servidor para definir.

schema Usado em configurações de banco de dados do `PostgreSQL` para especificar qual schema usar.

unix_socket Usado por drivers que o suportam para se conectar via arquivos de soquete Unix. Se você estiver usando o `PostgreSQL` e quiser usar os soquetes Unix, deixe a chave do host em branco.

ssl_key O caminho para o arquivo de chave SSL. (Somente suportado pelo `MySQL`).

ssl_cert O caminho para o arquivo de certificado SSL. (Somente suportado pelo `MySQL`).

ssl_ca O caminho do arquivo de autoridade de certificação SSL. (Somente suportado pelo `MySQL`).

init Uma lista de queries que devem ser enviadas para o servidor de banco de dados como quando a conexão é criada.

log Defina para `true` para habilitar o log de query. Quando habilitado, queries serão registradas(logged) em um nível debug com o escopo `“queriesLog”`.

quoteIdentifiers Defina para `true` se você estiver usando palavras reservadas os caracteres especiais nos nomes de suas tabelas ou colunas. Habilitando essa configuração, resultará em consultas criadas usando o [Construtor de queries](#) com identificadores citados (quoted) ao criar SQL. Deve ser notado, que isso diminui o desempenho porque cada consulta precisa ser percorrida e manipulada antes de ser executada.

flags Um array associativo de constantes `PDO` que devem ser passada para a instância `PDO` subjacente. Consulte a documentação do `PDO` sobre as flags suportadas pelo driver que você está usando.

cacheMetadata Tanto um boolean `true`, ou uma string contendo a configuração de cache para armazenar metadados. Desativar o cache de metadados não é a aconselhado e pode resultar em desempenho muito fraco. Consulte a seção [Metadata Caching](#) para obter mais informações.

Neste ponto, pode desejar dar uma olhada no *Convenções do CakePHP*. A correta nomenclatura para suas tables (e a adição de algumas colunas) podem garantir algumas funcionalidades gratuitas e ajudá-lo a evitar configuração. Por exemplo, se você nomear sua tabela de banco de dados `big_boxes`, sua tabela `BigBoxesTable` e o seu controller `BigBoxesController`, tudo funcionará em conjunto automaticamente. Por convenção, use sublinhados, minúsculas e plurais para os nomes de tabelas de banco de dados - por exemplo: `bakers`, `pastry_stores`, and `savory_cakes`.

Gerenciando Conexões

class `Cake\Datasource\ConnectionManager`

A classe *ConnectionManager* atua como um registro para acessar conexões de banco de dados que seu aplicativo tem. Ele fornece um lugar onde outros objetos podem obter referências às conexões existentes.

Acessando Conexões

static `Cake\Datasource\ConnectionManager::get($name)`

Uma vez configurada, as conexões podem ser obtidas usando `Cake\Datasource\ConnectionManager::get()`. Este método irá construir e carregar uma conexão se não tiver sido construído antes ou retornar a conexão conhecida existente:

```
use Cake\Datasource\ConnectionManager;

$conn = ConnectionManager::get('default');
```

Ao tentar carregar conexões que não existem será lançado uma exceção.

Criando Conexões em Tempo de Execução

Usando `config()` e `get()` você pode criar novas conexões que não estão definidas em seus arquivos de configuração em tempo de execução:

```
ConnectionManager::config('my_connection', $config);
$conn = ConnectionManager::get('my_connection');
```

Consulte a seção *Configuração* para mais informações sobre os dados de configuração usados ao criar conexões.

Tipos de Dados

class `Cake\Database\Type`

Como nem todos os fornecedores de banco de dados incluem o mesmo conjunto de tipos de dados, ou os mesmos nomes para tipos de dados semelhantes, o CakePHP fornece um conjunto de tipos de dados abstraídos para uso com a camada do banco de dados. Os tipos suportados pelo CakePHP são:

string Geralmente usado para colunas dos tipos CHAR ou VARCHAR. Ao usar a opção `fixed` forçará uma coluna CHAR. No SQL Server, os tipos NCHAR e NVARCHAR são usados.

text Mapeia para tipos de TEXT.

uuid Mapeia para o tipo UUID se um banco de dados fornecer um, caso contrário, isso gerará um campo CHAR(36).

integer Mapeia para o tipo INTEGER fornecido pelo banco de dados. O BIT ainda não é suportado neste momento.

biginteger Mapeia para o tipo BIGINT fornecido pelo banco de dados.

float Mapeia para DOUBLE ou FLOAT, dependendo do banco de dados. A opção `precision` pode ser usada para definir a precisão utilizada.

decimal Mapeia para o tipo DECIMAL. Suporta as opções `length` e `precision`.

boolean Mapeia para BOOLEAN, exceto no MySQL, onde TINYINT(1) é usado para representar booleans. BIT(1) ainda não é suportado neste momento.

binary Mapeia para os tipos BLOB ou BYTEA fornecido pelo banco de dados.

date Mapeia para o tipo de coluna DATE de fuso horário nativo. O valor de retorno desse tipo de coluna é `Cake\I18n\Date` que estende a classe nativa `DateTime`.

datetime Mapeia para o tipo de coluna DATETIME de fuso horário nativo. No PostgreSQL e no SQL Server, isso se transforma em um tipo TIMESTAMP. O valor de retorno padrão desse tipo de coluna é `Cake\I18n\Date` que estende a classe nativa `DateTime` e [Chronos](#)¹⁰⁶.

timestamp Mapeia para o tipo TIMESTAMP.

time Mapeia para um tipo TIME em todos bancos de dados.

json Mapeia para um tipo JSON se disponível, caso contrário mapeia para TEXT. O tipo 'json' foi adicionado na versão 3.3.0

Esses tipos são usados tanto nos recursos de schema reflection que o CakePHP fornece, quanto nos recursos de geração de schema que o CakePHP utiliza ao usar fixtures de testes.

Cada tipo também pode fornecer funções de tradução entre representações de PHP e SQL. Esses métodos são invocados com base nos type hints fornecidos ao fazer consultas. Por exemplo, uma coluna marcada como 'datetime' automaticamente converterá os parâmetros de entrada das instâncias `DateTime` em timestamp ou string de data formatada. Da mesma forma, as colunas 'binary' aceitarão manipuladores de arquivos e gerarão manipuladores de arquivos ao ler dados.

Adicionando Tipos Personalizados

static `Cake\Database\Type::map($name, $class)`

Se você precisa usar tipos específicos do fornecedor que não estão incorporados no CakePHP, você pode adicionar novos tipos adicionais ao sistema de tipos do CakePHP. As classes de tipos devem implementar os seguintes métodos:

- `toPHP`: Converte valor vindo do banco de dados em um tipo equivalente do PHP.
- `toDatabase`: Converte valor vindo do PHP em um tipo aceitável por um banco de dados.
- `toStatement`: Converte valor para seu equivalente Statement.
- `marshal`: Converte dados simples em objetos PHP.

Uma maneira fácil de atender a interface básica é estender `Cake\Database\Type`. Por exemplo, se quiséssemos adicionar um tipo JSON, poderíamos fazer a seguinte classe de tipo:

```
// in src/Database/Type/JsonType.php

namespace App\Database\Type;

use Cake\Database\Driver;
use Cake\Database\Type;
use PDO;

class JsonType extends Type
```

¹⁰⁶ <https://github.com/cakephp/chronos>

```

{

    public function toPHP($value, Driver $driver)
    {
        if ($value === null) {
            return null;
        }
        return json_decode($value, true);
    }

    public function marshal($value)
    {
        if (is_array($value) || $value === null) {
            return $value;
        }
        return json_decode($value, true);
    }

    public function toDatabase($value, Driver $driver)
    {
        return json_encode($value);
    }

    public function toStatement($value, Driver $driver)
    {
        if ($value === null) {
            return PDO::PARAM_NULL;
        }
        return PDO::PARAM_STR;
    }

}

```

Por padrão, o método `toStatement()` tratará os valores como strings que funcionarão para o nosso novo tipo. Uma vez que criamos nosso novo tipo, nós precisamos adicioná-lo ao mapeamento de tipo. Durante o bootstrap do nosso aplicativo, devemos fazer o seguinte:

```

use Cake\Database\Type;

Type::map('json', 'App\Database\Type\JsonType');

```

Novo na versão 3.3.0: A classe `JsonType` descrita neste exemplo foi adicionada ao core.

Nós podemos então sobrecarregar os dados de schema refletido para usar nosso novo tipo, e a camada de banco de dados do CakePHP converterá automaticamente nossos dados JSON ao criar consultas. Você pode usar os tipos personalizados que você criou mapeando os tipos no seu método `_initializeSchema()` da Tabela:

```

use Cake\Database\Schema\TableSchema;

class WidgetsTable extends Table
{
    protected function _initializeSchema(TableSchema $schema)
    {
        $schema->columnType('widget_prefs', 'json');
        return $schema;
    }
}

```

```
}
```

Mapeando Tipos de Dados Personalizados para Expressões SQL

Novo na versão 3.3.0: O suporte de mapeamento de tipos de dados personalizados para expressões SQL foi adicionado na versão 3.3.0.

O exemplo anterior mapeia um tipo de dados personalizado para um tipo de coluna 'json' que é facilmente representado como uma string em uma instrução SQL. Os tipos complexos de dados SQL não podem ser representados como strings/integers em consultas SQL. Ao trabalhar com esses tipos de dados, sua classe Type precisa implementar a interface `Cake\Database\Type\ExpressionTypeInterface`. Essa interface permite que seu tipo personalizado represente um valor como uma expressão SQL. Como exemplo, nós vamos construir uma simples classe Type para manipular dados do tipo POINT do MySQL. Primeiramente, vamos definir um objeto 'value' que podemos usar para representar dados POINT no PHP:

```
// in src/Database/Point.php
namespace App\Database;

// Our value object is immutable.
class Point
{
    protected $_lat;
    protected $_long;

    // Factory method.
    public static function parse($value)
    {
        // Parse the data from MySQL.
        return new static($value[0], $value[1]);
    }

    public function __construct($lat, $long)
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }

    public function long()
    {
        return $this->_long;
    }
}
```

Com o nosso objeto de valor criado, nós vamos precisar de uma classe Type para mapear dados nesse objeto de valor e em expressões SQL:

```
namespace App\Database\Type;

use App\Database\Point;
use Cake\Database\Expression\FunctionExpression;
use Cake\Database\Type as BaseType;
```

```

use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, Driver $d)
    {
        return Point::parse($value);
    }

    public function marshal($value)
    {
        if (is_string($value)) {
            $value = explode(',', $value);
        }
        if (is_array($value)) {
            return new Point($value[0], $value[1]);
        }
        return null;
    }

    public function toExpression($value)
    {
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
        // Handle other cases.
    }
}

```

A classe acima faz algumas coisas interessantes:

- O método `toPHP` lida com o parse de resultados de consulta SQL em um objeto de valor.
- O método `marshal` lida com a conversão, de dados como os dados de requisição, em nosso objeto de valor. Nós vamos aceitar valores string como `'10.24,12.34'` e array por enquanto.
- O método `toExpression` lida com a conversão do nosso objeto de valor para as expressões SQL equivalentes. No nosso exemplo, o SQL resultante seria algo como `POINT(10.24,12.34)`.

Uma vez que criamos nosso tipo personalizado, precisaremos *conectar nosso tipo personalizado à nossa classe table*.

Habilitando Objetos DateTime Imutáveis

Novo na versão 3.2: Immutable date/time objetos foram adicionados na versão 3.2.

Como objetos Date/Time são facilmente modificados, o CakePHP permite você habilitar objetos de valores imutáveis. Isso é melhor feito no arquivo **config/bootstrap.php** da sua aplicação:

```
Type::build('datetime')->useImmutable();
Type::build('date')->useImmutable();
Type::build('time')->useImmutable();
Type::build('timestamp')->useImmutable();
```

Nota: Novas aplicações terão objetos imutáveis habilitado por padrão.

Classes de Conexão

class Cake\Database\Connection

As classes de conexão fornecem uma interface simples para interagir com conexões de banco de dados de modo consistente. Elas servem como uma interface mais abstrata para a camada do driver e fornece recursos para executar consultas, logar (logging) consultas e realizar operações transacionais.

Executando Consultas

Cake\Database\Connection::query(\$sql)

Uma vez que você obteve um objeto de conexão, você provavelmente querará executar algumas consultas com ele. A camada de abstração de banco de dados do CakePHP fornece recursos de wrapper em cima do PDO e drivers nativos. Esses wrappers fornecem uma interface similar ao PDO. Há algumas formas diferentes de executar consultas, dependendo do tipo de consulta que você precisa executar e do tipo de resultados que você precisa receber. O método mais básico é o query() que lhe permite executar consultas SQL já prontas:

```
$stmt = $conn->query('UPDATE articles SET published = 1 WHERE id = 2');
```

Cake\Database\Connection::execute(\$sql, \$params, \$types)

O método query() não aceita parâmetros adicionais. Se você precisa de parâmetros adicionais, você deve usar o método execute(), que permite que placeholders sejam usados:

```
$stmt = $conn->execute(
    'UPDATE articles SET published = ? WHERE id = ?',
    [1, 2]
);
```

Sem qualquer informação de indução de tipo, execute assumirá que todos os placeholders são valores do tipo string. Se você precisa vincular tipos específicos de dados, você pode usar seus nomes de tipos abstratos ao criar uma consulta:

```
$stmt = $conn->execute(
    'UPDATE articles SET published_date = ? WHERE id = ?',
    [new DateTime('now'), 2],
    ['date', 'integer']
);
```

Cake\Database\Connection::newQuery()

Isso permite que você use tipos de dados ricos em suas aplicações e convertê-los adequadamente em instruções SQL. A última e mais flexível maneira de criar consultas é usar o *Construtor de queries*. Essa abordagem lhe permite criar consultas complexas e expressivas sem ter que usar SQL específico de plataforma:


```
$query = $conn->newQuery();
$query->update('articles')
    ->set(['published' => true])
    ->where(['id' => 2]);
$stmt = $query->execute();
```

Ao usar o construtor de consulta ([query builder](#)), nenhum SQL será enviado para o servidor do banco de dados até que o método `execute()` é chamado ou a consulta seja iterada. Iterar uma consulta irá primeiro executá-la e então começar a iterar sobre o conjunto de resultados:

```
$query = $conn->newQuery();
$query->select('*')
    ->from('articles')
    ->where(['published' => true]);

foreach ($query as $row) {
    // Faz alguma coisa com a linha.
}
```

Nota: Quando você tem uma instância de `Cake\ORM\Query` você pode usar o método `all()` para obter o conjunto de resultados de consultas `SELECT`.

Usando Transações

Os objetos de conexão lhe fornecem algumas maneiras simples de realizar transações de banco de dados. A maneira mais básica de fazer transações é através dos métodos `begin()`, `commit()` e `rollback()`, que mapeiam para seus equivalentes em SQL:

```
$conn->begin();
$conn->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
$conn->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
$conn->commit();
```

`Cake\Database\Connection::transactional(callable $callback)`

Além disso, essas instâncias de interface de conexão também fornecem o método `transactional()` que torna o tratamento das chamadas `begin/commit/rollback` muito mais simples:

```
$conn->transactional(function ($conn) {
    $conn->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
    $conn->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
});
```

Além de consultas básicas, você pode executar consultas mais complexas usando [Construtor de queries](#) ou [Objetos de tabela](#). O método `transactional` vai fazer o seguinte:

- Chamar método `begin`.
- Chamar a closure fornecida.
- Se a closure lançar uma exceção, um `rollback` será emitido. A exceção original será re-lançada.
- Se a closure retornar `false`, um `rollback` será emitido.
- Se a closure for executada com sucesso, a transação será cometida ([committed](#)).

Interagindo com Instruções

Ao usar a API do banco de dados de baixo nível, você muitas vezes encontrará objetos de instrução. Esses objetos lhe permitem manipular a instrução preparada subjacente do driver. Depois de criar e executar um objeto de consulta, ou usando `execute()` você terá uma instância `StatementDecorator`. Isso envolve o objeto de instrução básico subjacente e fornece alguns recursos adicionais.

Preparando um Instrução

Você pode criar um objeto de instrução usando `execute()` ou `prepare()`. O método `execute()` retorna uma instrução com os valores fornecidos ligados a ela. Enquanto que o `prepare()` retorna uma instrução incompleta:

```
// Instruções do ``execute`` terão valores já vinculados a eles.
$stmt = $conn->execute(
    'SELECT * FROM articles WHERE published = ?',
    [true]
);

// Instruções do ``prepare`` serão parâmetros para placeholders.
// Você precisa vincular os parâmetros antes de executar.
$stmt = $conn->prepare('SELECT * FROM articles WHERE published = ?');
```

Uma vez que você preparou uma instrução, você pode vincular dados adicionais e executá-lo.

Binding Values

Uma vez que você criou uma instrução preparada, você talvez precise vincular dados adicionais. Você pode vincular vários valores ao mesmo tempo usando o método `bind()`, ou vincular elementos individuais usando `bindValue()`:

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = ? AND created > ?'
);
// Vincular vários valores
$stmt->bind(
    [true, new DateTime('2013-01-01')],
    ['boolean', 'date']
);

// Vincular único valor
$stmt->bindValue(1, true, 'boolean');
$stmt->bindValue(2, new DateTime('2013-01-01'), 'date');
```

Ao criar instruções, você também pode usar chave de array nomeadas em vez de posicionais:

```
$stmt = $conn->prepare(
    'SELECT * FROM articles WHERE published = :published AND created > :created'
);

// Vincular vários valores
$stmt->bind(
    ['published' => true, 'created' => new DateTime('2013-01-01')],
    ['published' => 'boolean', 'created' => 'date']
);

// Vincular um valor único
```

```
$stmt->bindValue('published', true, 'boolean');  
$stmt->bindValue('created', new DateTime('2013-01-01'), 'date');
```

Aviso: Você não pode misturar posicionais e chave de array nomeadas na mesma instrução.

Executando & Obtendo Linhas

Depois de preparar uma instrução e vincular dados a ela, você pode executá-la e obter linhas. As instruções devem ser executadas usando o método `execute()`. Uma vez executado, os resultados podem ser obtidos usando `fetch()`, `fetchAll()` ou iterando a instrução:

```
$stmt->execute();  
  
// Lê uma linha.  
$row = $stmt->fetch('assoc');  
  
// Lê todas as linhas.  
$rows = $stmt->fetchAll('assoc');  
  
// Lê linhas através de iteração.  
foreach ($rows as $row) {  
    // Do work  
}
```

Nota: Lendo linhas através de iteração irá obter linhas no modo ‘both’. Isso significa que você obterá os resultados indexados numericamente e indexados associativamente.

Obtendo Contagens de Linha

Depois de executar uma declaração, você pode buscar o número de linhas afetadas:

```
$rowCount = count($stmt);  
$rowCount = $stmt->rowCount();
```

Verificando Códigos de Erro

Se a sua consulta não foi bem sucedida, você pode obter informações de erro relacionadas usando os métodos `errorCode()` e `errorInfo()`. Estes métodos funcionam da mesma maneira que os fornecidos pelo PDO:

```
$code = $stmt->errorCode();  
$info = $stmt->errorInfo();
```

Log de Consultas

O log de consultas pode ser habilitado ao configurar sua conexão definindo a opção `log` com o valor `true`. Você também pode alternar o log de consulta em tempo de execução, usando o método `logQueries`:

```
// Habilita log de consultas.  
$conn->logQueries(true);  
  
// Desabilita o log de consultas.  
$conn->logQueries(false);
```

Quando o log de consultas está habilitado, as consultas serão logadas em `Cake\Log\Log` usando o nível `'debug'`, e o escopo `'queriesLog'`. Você precisará ter um logger configurado para capturar esse nível e escopo. Logar no `stderr` pode ser útil quando se estiver trabalhando com testes de unidade e logar em arquivos/syslog pode ser útil ao trabalhar com requisições web:

```
use Cake\Log\Log;  
  
// Console logging  
Log::config('queries', [  
    'className' => 'Console',  
    'stream' => 'php://stderr',  
    'scopes' => ['queriesLog']  
]);  
  
// File logging  
Log::config('queries', [  
    'className' => 'File',  
    'path' => LOGS,  
    'file' => 'queries.log',  
    'scopes' => ['queriesLog']  
]);
```

Nota: Log de consultas destina-se apenas para usos de depuração/desenvolvimento. Você nunca deve habilitar o log de consultas em ambiente de produção, pois isso afetará negativamente o desempenho de sua aplicação.

Identifier Quoting

Por padrão, o CakePHP **não** cita (`quote`) identificadores em consultas SQL geradas. A razão disso é que a citação de identificadores tem algumas desvantagens:

- Sobrecarga de desempenho - Citar identificadores é muito mais lentos e complexos do que não fazê-lo.
- Não é necessário na maioria dos casos - Em bancos de dados não legados que seguem as convenções do CakePHP não há motivo para citar identificadores.

Se você estiver usando um schema legado que requer citação de identificador, você pode habilitar isso usando a configuração `quoteIdentifiers`` em seu *Configuração*. Você também pode habilitar esse recurso em tempo de execução:

```
$conn->driver()->autoQuoting(true);
```

Quando habilitado, a citação de identificador causará uma `traversal query` adicional que converte todos os identificadores em objetos `IdentifierExpression`.

Nota: Os fragmentos de SQL contidos em objetos `QueryExpression` não serão modificados.

Metadata Caching

O ORM do CakePHP usa reflexão de banco de dados para determinar a schema, índices e chaves estrangeiras que sua aplicação contém. Como esse metadado é alterado com pouca frequência e pode ser caro de acessar, ele geralmente é armazenado em cache. Por padrão, os metadados são armazenados na configuração de cache `_cake_model_`. Você pode definir uma configuração de cache personalizada usando a opção `cacheMetadata` na sua configuração de `datasource`:

```
'Datasources' => [
    'default' => [
        // Other keys go here.

        // Use the 'orm_metadata' cache config for metadata.
        'cacheMetadata' => 'orm_metadata',
    ]
],
```

Você também pode configurar o cache de metadados em tempo de execução com o método `cacheMetadata()`:

```
// Desabilitar o cache
$connection->cacheMetadata(false);

// Habilitar tohe cache
$connection->cacheMetadata(true);

// Utilizar uma configuração de cache personalizada
$connection->cacheMetadata('orm_metadata');
```

O CakePHP também inclui uma ferramenta CLI para gerenciar caches de metadados. Confira o capítulo *ORM Cache Shell* para obter mais informações.

Criando Banco de Dados

Se você quer criar uma conexão sem selecionar um banco de dados, você pode omitir o nome do banco de dados:

```
$dsn = 'mysql://root:password@localhost/';
```

Agora você pode usar seu objeto de conexão para executar consultas que cria/modifica bancos de dados. Por exemplo, para criar um banco de dados:

```
$connection->query("CREATE DATABASE IF NOT EXISTS my_database");
```

Nota: Ao criar um banco de dados, é uma boa idéia definir o conjunto de caracteres e os parâmetros de collation. Se esses valores estiverem faltando, o banco de dados definirá quaisquer valores padrão de sistema que ele use.

Construtor de queries

Returning the Total Count of Records

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Atualizando Dados

Objetos de tabela

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁰⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Lifecycle Callbacks

Behaviors

addBehavior (\$name, array \$options = [])

Behaviors fornecem uma maneira fácil de criar partes de lógica horizontalmente reutilizáveis relacionadas às classes de tabela. Você pode estar se perguntando por que os behaviors são classes regulares e não traits. O principal motivo para isso é event listeners. Enquanto as traits permitiriam partes reutilizáveis de lógica, eles complicariam o uso de eventos.

Para adicionar um behavior à sua tabela, você pode chamar o método `addBehavior()`. Geralmente o melhor lugar para fazer isso é no método `initialize()`:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }
}
```

Como acontece com as associações, você pode usar *syntaxe plugin* e fornecer opções de configuração adicionais:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
```

¹⁰⁷ <https://github.com/cakephp/docs>

¹⁰⁸ <https://github.com/cakephp/docs>

```

{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}

```

Você pode descobrir mais sobre behavior, incluindo os behaviors fornecidos pelo CakePHP no capítulo sobre *Behaviors (Comportamentos)*.

Configurando Conexões

Entidades

class Cake\ORM\Entity

Enquanto *Objetos de tabela* representam e fornecem acesso a uma coleção de objetos, entidades representam linhas individuais ou objetos de domínio na sua aplicação. Entidades contêm propriedades persistentes e métodos para manipular e acessar os dados que elas contêm.

Entidades são criadas para você pelo CakePHP cada vez que utilizar o `find()` em um objeto de Table.

Criando Classes de Entidade

Você não precisa criar classes de entidade para iniciar com o ORM no CakePHP. No entanto, se você deseja ter lógica personalizada nas suas entidades, você precisará criar classes. Por convenção, classes de entidades ficam em **src/Model/Entity/**. Se a nossa aplicação tem um tabela `articles`, poderíamos criar a seguinte entidade:

```

// src/Model/Entity/Article.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
}

```

Neste momento, essa entidade não faz muita coisa. No entanto, quando carregarmos dados da nossa tabela `articles`, obteremos instâncias dessa classe.

Nota: Se você não definir uma classe de entidade o CakePHP usará a classe Entity básica.

Criando Entidade

Entidades podem ser instanciadas diretamente:

```
use App\Model\Entity\Article;

$article = new Article();
```

Ao instanciar uma entidade, você pode passar as propriedades com os dados que deseja armazenar nelas:

```
use App\Model\Entity\Article;

$article = new Article([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

Outra maneira de obter novas entidades é usando o método `newEntity()` dos objetos `Table`:

```
use Cake\ORM\TableRegistry;

$article = TableRegistry::get('Articles')->newEntity();
$article = TableRegistry::get('Articles')->newEntity([
    'id' => 1,
    'title' => 'New Article',
    'created' => new DateTime('now')
]);
```

Acessando Dados de Entidade

Entidades fornecem algumas maneiras de acessar os dados que contêm. Normalmente, você acessará os dados de uma entidade usando notação de objeto (object notation):

```
use App\Model\Entity\Article;

$article = new Article;
$article->title = 'This is my first post';
echo $article->title;
```

Você também pode usar os métodos `get()` e `set()`:

```
$article->set('title', 'This is my first post');
echo $article->get('title');
```

Ao usar `set()`, você pode atualizar várias propriedades ao mesmo tempo usando um array:

```
$article->set([
    'title' => 'My first post',
    'body' => 'It is the best ever!'
]);
```

Aviso: Ao atualizar entidades com dados de requisição, você deve especificar com `whitelist` quais campos podem ser definidos com atribuição de massa.

Accessors & Mutators

Além da simples interface `get/set`, as entidades permitem que você forneça métodos acessadores e mutadores. Esses métodos deixam você personalizar como as propriedades são lidas ou definidas.

Acessadores usam a convenção de `_get` seguido da versão CamelCased do nome do campo.

```
Cake\ORM\Entity::get($field)
```

Eles recebem o valor básico armazenado no array `_properties` como seu único argumento. Acessadores serão usadas ao salvar entidades, então seja cuidadoso ao definir métodos que formatam dados, já que os dados formatados serão persistido. Por exemplo:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected function _getTitle($title)
    {
        return ucwords($title);
    }
}
```

O acessador seria executado ao obter a propriedade através de qualquer uma dessas duas formas:

```
echo $user->title;
echo $user->get('title');
```

Você pode personalizar como as propriedades são atribuídas definindo um mutador:

```
Cake\ORM\Entity::set($field = null, $value = null)
```

Os métodos mutadores sempre devem retornar o valor que deve ser armazenado na propriedade. Como você pode ver acima, você também pode usar mutadores para atribuir outras propriedades calculadas. Ao fazer isso, seja cuidadoso para não introduzir nenhum loops, já que o CakePHP não impedirá os métodos mutadores de looping infinitos.

Os mutadores permitem você converter as propriedades conforme são atribuídas, ou criar dados calculados. Os mutadores e acessores são aplicados quando as propriedades são lidas usando notação de objeto (object notation), ou usando os métodos `get()` e `set()`. Por exemplo:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use Cake\Utility\Text;

class Article extends Entity
{
    protected function _setTitle($title)
    {
        return Text::slug($title);
    }
}
```

O mutador seria executado ao atribuir a propriedade através de qualquer uma dessas duas formas:

```
$user->title = 'foo'; // slug is set as well
$user->set('title', 'foo'); // slug is set as well
```

Criando Propriedades Virtuais

Ao definir acessadores, você pode fornecer acesso aos campos/propriedades que não existem. Por exemplo, se sua tabela users tem first_name e last_name, você poderia criar um método para o full_name:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected function _getFullName()
    {
        return $this->_properties['first_name'] . ' ' .
            $this->_properties['last_name'];
    }
}
```

Você pode acessar propriedades virtuais como se elas existissem na entidade. O nome da propriedade será a versão lower case e underscored do método:

```
echo $user->full_name;
```

Tenha em mente que as propriedades virtuais não podem ser usadas nos finds. Se você deseja que as propriedades virtuais façam parte de representações JSON ou array de suas entidades, consulte [Expondo Propriedades Virtuais](#).

Verificando se uma Entidade Foi Modificada

```
Cake\ORM\Entity::dirty($field = null, $dirty = null)
```

Você pode querer fazer código condicional com base em se as propriedades foram modificadas ou não em uma entidade. Por exemplo, você pode só querer validar campos quando eles mudarem:

```
// See if the title has been modified.
$article->dirty('title');
```

Você também pode marcar campos como sendo modificados. Isso é útil quando adiciona item em propriedades do tipo array:

```
// Adiciona um comentário e marca o campo como modificado
$article->comments[] = $newComment;
$article->dirty('comments', true);
```

Além disso, você também pode basear o seu código condicional nos valores de propriedades originais usando o método getOriginal(). Esse método retornará o valor original da propriedade se tiver sido modificado ou seu valor real.

Você também pode verificar se há mudanças em qualquer propriedade na entidade:

```
// Verifica se a entidade foi modificada
$article->dirty();
```

Para remover a marca de modificação (dirty flag) em um entidade, você pode usar o método `clean()`:

```
$article->clean();
```

Ao criar uma nova entidade, você pode evitar que os campos sejam marcados como modificados (dirty) passando uma opção extra:

```
$article = new Article(['title' => 'New Article'], ['markClean' => true]);
```

Para obter uma lista de todos as propriedades modificada (dirty) de uma Entity, você pode chamar:

```
$dirtyFields = $entity->getDirty();
```

Novo na versão 3.4.3: `getDirty()` foi adicionado.

Erros de Validação

`Cake\ORM\Entity::errors($field = null, $errors = null)`

Depois que você *salva uma entidade*, quaisquer erros de validação serão armazenados na própria entidade. Você pode acessar os erros de validação usando os métodos `getErrors()` ou `getError()`:

```
// Obtem todos os erros
$errors = $user->getErrors();
// Antes da versão 3.4.0
$errors = $user->errors();

// Obtem os erros para um único campo.
$errors = $user->getError('password');
// Antes da versão 3.4.0
$errors = $user->errors('password');
```

Os métodos `setErrors()` ou `setError()` podem também ser usados para definir erros em uma entidade, tornando mais fácil testar código que trabalha com mensagens de erro:

```
$user->setError('password', ['Password is required']);
$user->setErrors(['password' => ['Password is required'], 'username' => ['Username is_
↪required']]);
// Antes da versão 3.4.0
$user->errors('password', ['Password is required.']);
```

Atribuição em Massa

Embora a definição de propriedades para entidades em massa seja simples e conveniente, isso pode criar problemas de segurança significativos. Atribuindo em massa dados de usuário apartir da requisição a uma entidade permite ao usuário modificar todas e quaisquer colunas. Ao usar classes de entidade anônimas ou criar a classe de entidade com *Bake Console*, o CakePHP não protege contra a atribuição em massa.

A propriedade `_accessible` permite que você forneça um mapa de propriedades e se elas podem ou não ser atribuídas em massa. Os valores `true` e `false` indicam se um campo pode ou não ser atribuído em massa:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
```

```
{  
    protected $_accessible = [  
        'title' => true,  
        'body' => true  
    ];  
}
```

Além dos campos concretos, existe um campo especial `*` que define o comportamento de fallback se um campo não for especificamente nomeado:

```
namespace App\Model\Entity;  
  
use Cake\ORM\Entity;  
  
class Article extends Entity  
{  
    protected $_accessible = [  
        'title' => true,  
        'body' => true,  
        '*' => false,  
    ];  
}
```

Nota: Se a propriedade `*` não for definida, seu padrão será `false`.

Evitando Proteção de Atribuição em Massa

Ao criar uma nova entidade usando a palavra-chave `new`, você pode dizer para não se proteger de atribuição em massa:

```
use App\Model\Entity\Article;  
  
$article = new Article(['id' => 1, 'title' => 'Foo'], ['guard' => false]);
```

Modificando os Campos Vigiaados em Tempo de Execução

Você pode modificar a lista de campos vigiaados em tempo de execução usando o método `accessible`:

```
// Faz user_id ser acessível.  
$article->accessible('user_id', true);  
  
// Faz title ser vigiado.  
$article->accessible('title', false);
```

Nota: A modificação de campos afetam apenas a instância em que o método é chamado.

Ao usar os métodos `newEntity()` e `patchEntity()` nos objetos `Table`, você pode personalizar a proteção de atribuição em massa com opções, Por favor consulte a seção [Alterando Campos Acessíveis](#) para obter mais informações.

Ignorando Proteção de Campo

Existem algumas situações em que você deseja permitir atribuição em massa para campos vigiados (guarded):

```
$article->set($properties, ['guard' => false]);
```

Definindo a opção `guard` como `false`, você pode ignorar a lista de campos acessíveis para uma única chamado ao método `set()`.

Verificando se uma Entidade foi Persistida

Frequentemente é necessário saber se uma entidade representa uma linha que já está no banco de dados. Nessas situações, use o método `isNew()`:

```
if (!$article->isNew()) {
    echo 'This article was saved already!';
}
```

Se você está certo que uma entidade já foi persistida, você pode usar `isNew()` como um setter:

```
$article->isNew(false);

$article->isNew(true);
```

Lazy Loading Associations

Embora que eager loading de associações é geralmente o modo mais eficiente de acessar suas associações, pode existir momentos em que você precisa carregar seus dados sobre demanda (lazy load). Antes de entrar em como carregar associações sobre demanda, devemos discutir as diferenças entre eager loading e lazy loading de associações:

Eager loading Eager loading utiliza joins (onde possível) para buscar os dados do banco de dados em poucas consultas possível. Quando uma consulta separada é necessária, como no caso de uma associação HasMany, uma única consulta é emitida para buscar *todos* os dados associados para o conjunto atual de objetos.

Lazy loading Lazy loading difere o carregamento de associação até que seja absolutamente necessário. Embora isso possa economizar tempo de CPU, porque possivelmente dados não utilizados não são hidratados (hydrated) em objetos, isso pode resultar em muitas outras consultas sendo emitidas para o banco de dados. Por exemplo, fazer um loop sobre um conjunto de artigos e seus comentários frequentemente emitirão N consultas onde N é o número de artigos sendo iterados.

Embora lazy loading não esteja incluído no ORM do CakePHP, você pode usar um dos plugins da comunidade para fazer isso. Nós recomendamos o [LazyLoad Plugin](https://github.com/jeremyharris/cakephp-lazyload)¹⁰⁹

Depois de adicionar o plugin em sua entidade, você será capaz de fazer o seguinte:

```
$article = $this->Articles->findById($id);

// A propriedade comments foi carregado sobre demanda (lazy loaded)
foreach ($article->comments as $comment) {
    echo $comment->body;
}
```

¹⁰⁹ <https://github.com/jeremyharris/cakephp-lazyload>

Criando Código Re-utilizável com Traits

Você pode encontrar-se precisando da mesma lógica em várias classes de entidades. As Traits do PHP são perfeitas para isso. Você pode colocar as traits da sua aplicação em **src/Model/Entity**. Por convenção traits no CakePHP são sufixadas com `Trait` para que elas possam ser discerníveis de classes ou interfaces. Traits são geralmente um bom complemento para os behaviors, permitindo que você forneça funcionalidade para objetos de tabela e entidade.

Por exemplo, se tivéssemos plugin `SoftDeletable`, isso poderia fornecer uma trait. Essa trait poderia fornecer métodos para marcar entidades como `'deleted'`, o método `softDelete` poderia ser fornecido por uma trait:

```
// SoftDelete/Model/Entity/SoftDeleteTrait.php

namespace SoftDelete\Model\Entity;

trait SoftDeleteTrait
{
    public function softDelete()
    {
        $this->set('deleted', true);
    }
}
```

Você poderia então usar essa trait na sua classe de entidade importando-a e incluíndo-a:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;
use SoftDelete\Model\Entity\SoftDeleteTrait;

class Article extends Entity
{
    use SoftDeleteTrait;
}
```

Convertendo para Arrays/JSON

Ao construir APIs, você geralmente pode precisar converter entidades em arrays ou dados JSON. CakePHP torna isso simples:

```
// Obtem um array.
// Associações serão convertida com toArray() também.
$array = $user->toArray();

// Converte para JSON
// Associações serão convertida com jsonSerialize hook também.
$json = json_encode($user);
```

Ao converter uma entidade para um JSON, as listas de campos virtuais e ocultos são aplicadas. Entidades são recursivamente convertidas para JSON também. Isso significa que, se você eager loaded entidades e suas associações, o CakePHP manipulará corretamente a conversão dos dados associados no formato correto.

Expondo Propriedades Virtuais

Por padrão, campos virtuais não são exportados ao converter entidades para arrays ou JSON. Para expor propriedades virtuais, você precisa torná-las visíveis. Ao definir sua classe de entidade, você pode fornecer uma lista de propriedades virtuais que devem ser expostas:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected $_virtual = ['full_name'];
}
```

Esta lista pode ser modificada em tempo de execução usando o método `virtualProperties`:

```
$user->virtualProperties(['full_name', 'is_admin']);
```

Ocultando Propriedades

Muitas vezes, há campos que você não deseja ser exportado em formatos de array ou JSON. Por exemplo geralmente não é sensato expor hash de senha ou perguntas de recuperação de conta. Ao definir uma classe de entidade, defina quais propriedades devem ser ocultadas:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class User extends Entity
{
    protected $_hidden = ['password'];
}
```

Esta lista pode ser modificada em tempo de execução usando o método `hiddenProperties`:

```
$user->hiddenProperties(['password', 'recovery_question']);
```

Armazenando Tipos Complexos

Métodos Acessores & Mutadores em entidades não são destinados para conter a lógica de serializar e deserializar dados complexos vindo do banco de dados. Consulte a seção *Salvando Tipos Complexos (Complex Types)* para entender como sua aplicação pode armazenar tipos de dado complexos, como arrays e objetos.

Retornando dados e conjuntos de resultados

```
class Cake\ORM\Table
```

Quando executar uma query, você obterá um objeto Entidade. Nesta sessão discutiremos diferentes caminhos para se obter: entidades, carregar informações relacionais, abstratas, ou complexo relacional. Você poderá ler mais sobre *Entidades* ('Entity' em inglês).

Depurando Queries e Resultados

Quando o ORM foi implementado, era muito difícil depurar os resultados obtidos nas versões anteriores do CakePHP. Agora existem muitas formas fáceis de inspecionar os dados retornados pelo ORM.

- `debug($query)` Mostra o SQL e os parâmetros incluídos, não mostra resultados.
- `debug($query->all())` Mostra a propriedade ResultSet retornado pelo ORM.
- `debug($query->toArray())` Um caminho mais fácil para mostrar todos os resultados.
- `debug(json_encode($query, JSON_PRETTY_PRINT))` Exemplo em JSON.
- `debug($query->first())` Primeiro resultado obtido na query.
- `debug((string)$query->first())` Mostra as propriedades de uma única entidade em JSON.

Tente isto na camada Controller: `debug($this->{EntidadeNome}->find()->all());`

Pegando uma entidade com a chave primária

`Cake\ORM\Table::get($id, $options = [])`

Sempre que é necessário editar ou visualizar uma entidade ou dados relacionais você pode usar `get()`:

```
// No controller ou table tente isto.

// Retorna um único artigo pelo id primário.
$article = $articles->get($id);

// Retorna um artigo com seus comentários
$article = $articles->get($id, [
    'contain' => ['Comments']
]);
```

Quando não conseguir obter um resultado `Cake\Datasource\Exception\RecordNotFoundException` será disparado. Você poderá tratar esta exceção, ou converter num erro 404.

O método `find()` usa uma cache integrado. Você pode uma a opção `cache` quando chamar `get()` para uma performance na leitura - caching:

```
// No controller ou table tente isto.

// Use uma configuração de cache ou uma instância do CacheEngine do Cake com uma ID_
↳gerada.
$article = $articles->get($id, [
    'cache' => 'custom',
]);

// mykey reserva uma id especifica para determinado cache de resultados.
$article = $articles->get($id, [
    'cache' => 'custom', 'key' => 'mykey'
]);

// Desabilitando cache explicitamente
```



```
$article = $articles->get($id, [
    'cache' => false
]);
```

Por padrão o CakePHP possui um sistema interno de cache que viabiliza busca e aumenta a performance – não é recomendado desabilitar.

Opcionalmente você pode usar `get()` nas entidades com busca customizável *Personalizando Métodos de Consulta*. Por exemplo, você pode querer pegar todas as traduções de uma entidade. Poderá usar a opção `finder`:

```
$article = $articles->get($id, [
    'finder' => 'translations',
]);
```

Usando 'find()' para carregar dados

```
Cake\ORM\Table::find($type, $options = [])
```

Agora que você sabe e pode trabalhar com entidades, Precisarás carregá-las e gostarás muito como fazer isso. O caminho mais simples para carregar uma Entidade ou objetos relacionais método `find()`. `find` provê um extensível e fácil caminho para procurar e retornar dados, talvez você se interesse por in:

```
// No controller ou table.

// Procure todos os artigos
$query = $articles->find('all');
```

O valor retornado por qualquer método `find()` será sempre um `Cake\ORM\Query` objeto. A class `Query` assim permitindo que possa posteriormente refinar a consulta depois de criá-la. Objeto `Query` não será executado até que inicie uma busca por linhas, seja convertido num array, ou chamado outro método, exemplo: `all()`:

```
// No controller ou table.

// Retorne todos os artigos
// Até este ponto, nada acontece.
$query = $articles->find('all');

// Uma iteração executa a consulta
foreach ($query as $row) {
}

// Chamando all() executa a consulta.
// e retorna os conjuntos de resultados.
$results = $query->all();

// Linhas são retornadas em forma de array
$data = $results->toArray();

// Armazenando a consulta num array
$results = $query->toArray();
```

Nota: Você já sabe executar uma consulta, gostará de *Construtor de queries* para implementar e construir consultas otimizadas ou complexas, adicionando condições específicas, limites, incluindo associação ou uma interface mais fluente, ou busca de resultados por id de usuário logado.

```
// No controller ou table.
$query = $articles->find('all')
    ->where(['Articles.created >' => new DateTime('-10 days')])
    ->contain(['Comments', 'Authors'])
    ->limit(10);
```

Não se limite, poderá ir muito além com `find()`. Isto o ajuda com métodos simulados:

```
// No controller ou table.
$query = $articles->find('all', [
    'conditions' => ['Articles.created >' => new DateTime('-10 days')],
    'contain' => ['Authors', 'Comments'],
    'limit' => 10
]);
//Ao buscar todos os artigos, retorne somente artigos com data de hoje - 10 dias atrás
//Depois junto com esses artigos me retorne também seus autores e comentários.
↳inclusos.
```

Opções suportadas por `find()` são:

- `conditions` provê acesso direto na cláusula Where.
- `limit` Limite o número de resultados.
- `offset` Uma página que você quer. Use `page` para cálculo simplificado.
- `contain` defina uma associação para carregar.
- `fields` Quais campos você deseja carregar somente? Quando carregar somente alguns campos o lembre-se dos plugins, callbacks.
- `group` adicione um GROUP BY. muito usado para funções agregadas.
- `having` adicionar HAVING.
- `join` Defina um Join específico.
- `order` Ordenar resultados por.

Outras opções fora dessa lista, serão passadas para o `beforeFind` ou outras funções de tratamento, onde podem ser usados para tratar a consulta a sua maneira. Pode usar o método `getOptions()` no objeto para retornar as opções utilizadas. Quando uma consulta for passada para o controller, recomendamos uma leitura sobre consultas personalizadas em *Personalizando Métodos de Consulta*. Usando métodos de consultas personalizados, você terá um melhor reuso de seu código, e ficará fácil para testar a sua maneira.

Por padrão consultas retornam *Entidades* objeto. Você pode retorna array basico usando hydration:

```
$query->hydrate(false);

// $data is ResultSet that contains array data.
$data = $query->all();
```

Primeiro Resultado

O método `first()` permite pegar apenas o primeiro resultado da consulta. Caso não seja bem executado a cláusula `LIMIT 1` será aplicada:

```
// No controller ou table.
$query = $articles->find('all', [
    'order' => ['Articles.created' => 'DESC']
```

```
]);
$row = $query->first();
//Ex: Retorne todos os artigos, mais quero somente o primeiro.
```

Uma abordagem diferente `find('first')` da versão anterior do CakePHP. Você também pode usar o método `get()` caso queira carregar uma entidade pelo chave primária.

Nota: O método `first()` retorna null caso nenhum resultado seja encontrado.

Contando os resultados

Criando uma consulta você gosta do método `count()` para retornar a quantidade de resultados encontrado:

```
// No controller ou table.
$query = $articles->find('all', [
    'conditions' => ['Articles.title LIKE' => '%Ovens%']
]);
$number = $query->count();
//Retorne todos os artigos, me mostre quantos são.
```

Veja *Returning the Total Count of Records* para modos de uso diferentes com o método `count()`.

Encontrando Chaves/Pares de Valores

Frequentemente precisamos gerar um dados associados em array de nossas aplicações. Muito usado para criar o elemento `<select>`. O Cake provê um método simples e fácil `'lists'`:

```
// No controller ou table.
$query = $articles->find('list');
$data = $query->toArray();

// Os dados organizados :D
$data = [
    1 => 'First post',
    2 => 'Second article I wrote',
];
```

Com as opções adicionais as chaves de `$data` podem representar uma coluna de sua tabela, Por exemplo, use `'displayField()'` no objeto tabela na função `'initialize()'`, isto configura um valor a ser mostrado na chave:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->displayField('title');
    }
}
```

Quando se chama `list` você pode configurar quais campos deseja usar para a chave e valor passando as opções `keyField` e `valueField` respectivamente:

```
// No controller ou table.
$query = $articles->find('list', [
    'keyField' => 'slug',
    'valueField' => 'title'
]);
$data = $query->toArray();

// Dados organizados :D
$data = [
    'first-post' => 'First post',
    'second-article-i-wrote' => 'Second article I wrote',
];
//slug passa a ser a chave
// title o valor do option no select
```

Resultados podem ser agrupados se necessitar. Muito usado quando desejar diferencias Chave/Valores por grupo no elemento <optgroup> com FormHelper:

```
// No controller ou table
$query = $articles->find('list', [
    'keyField' => 'slug',
    'valueField' => 'title',
    'groupField' => 'author_id'
]);
$data = $query->toArray();

// Dados organizados :D
$data = [
    1 => [
        'first-post' => 'First post',
        'second-article-i-wrote' => 'Second article I wrote',
    ],
    2 => [
        // More data.
    ]
];
// Temos então os artigos com sua Chave/Valores diferenciados por autores.
```

Não é complicado, use dados associados e poderá gostar do resultado:

```
$query = $articles->find('list', [
    'keyField' => 'id',
    'valueField' => 'author.name'
])->contain(['Authors']);
//Retorne uma lista de todos os artigos, o id representará a identificação do artigo,
↳ porém seu valor será o nome do seu Author.
//Importante, sempre que pesquisar ou informar campos adicionais use o '.' como
↳ mostrado em 'valueField'.
```

Por ultimo, é muito bom quando podemos usar métodos criados em nossas entidades, isto também é possível no método 'list'. . Neste exemplo mostra o uso método mutador `_getFullName()` criado na entidade Author.

```
$query = $articles->find('list', [
    'keyField' => 'id',
    'valueField' => function ($e) {
        return $e->author->get('full_name');
    }
]);
```

```
//O valor da chave, representará o nome completo
//Que usa de uma função para acessar o método mutador criado na entidade
//Onde ao juntar o 1 nome com o 2 formará o nome completo.
```

Encontrando dados enfileirados

O método `find('threaded')` retorna que estarão relacionados por chaves. Por padrão o Cake usa o campo `parent_id`. Nesse modelo, é possível encontrar valores no banco de dados adjacentes. Todas as entidades correspondentes recebem um `parent_id` e são alocadas no atributo `children`:

```
// No controller ou table.
$query = $comments->find('threaded');

// Expandindo os comentários de outros comentários
$query = $comments->find('threaded', [
    'keyField' => $comments->primaryKey(),
    'parentField' => 'parent_id'
]);
$results = $query->toArray();
// transformando todos os resultados em array.

echo count($results[0]->children);
//Para o primeiro resultado, mostra quantos filhos possui ou registros relacionados e
↳co-relacionados.
echo $results[0]->children[0]->comment;
//Mostre o comentário relacionado ao primeiro comentário
```

Um pouco mal explicado pela equipe do Cake, quando buscamos por dados enfileirados podemos ir bem além, até perceber que pode se encaixar perfeitamente em uma carrinho de shopping com seus itens e quantidades co-relacionados. O `parentField` e `keyField` chaves que serão usadas para encontrar ocorrências.

Será mais interessante quando aprender sobre árvore de dados ao considerar *Tree* posteriormente.

Personalizando Métodos de Consulta

Mostramos os exemplos de uso do `all` e `list`. Ficará interessado em saber as inúmeras possibilidades, e que também recomendamos seriamente, que você as implemente. Um método personalizado de busca pode ser ideal para simplificar processos, consultar dados complexos, otimizar buscas, ou criar uma busca padrão em um método simplificado feito por você. Eles podem ser definidos na criação do objeto tabela e devem obedecer a convenção padrão do Cake. Ao criar um método deverá iniciar seu nome com `find` e logo após adicionar o nome desejado para sua busca personalizada, exemplo: `find` e adicionar `Users = findUsers`. É de grande ajuda, por exemplo, quando queremos que em uma busca, nossa consulta sempre tenha a condição de que seus resultados sejam de um determinado usuário, ou que em um carrinho tenha sua própria listra agregada, sem precisar encher o controller de códigos e facilitando muito a manutenção no reuso de código. Neste exemplo mostramos como encontrarmos um artigo quando este estiver publicado somente.:

```
use Cake\ORM\Query;
use Cake\ORM\Table;

//Lembre se, deverá cria-lo no objeto Artigos
//Ou melhor /src/Model/Table/ArticlesTable.php

class ArticlesTable extends Table
{
```

```
//Nosso método personalizado
public function findOwnedBy(Query $query, array $options)
{
    $user = $options['user'];
    return $query->where(['author_id' => $user->id]);
}

}

// No controller ou table.
$articles = TableRegistry::get('Articles');
$query = $articles->find('ownedBy', ['user' => $userEntity]);
//Retorne todos os artigos, quero que seja de meu usuário, porém somente os já
↪publicados.
```

O método traz muita funcionalidade, em alguns casos precisamos definir uma pilha de lógica, isto será possível usando o atributo `$options` para personalização de consulta com lógica irrelevante. Sem esforço você pode expressar algumas consultas complexas. Assumindo que você tem ambas as buscas ‘published’ e ‘recent’, poderia fazer assim:

```
// No controller ou table.
$articles = TableRegistry::get('Articles');
$query = $articles->find('published')->find('recent');
//Busque todos os artigos, dentre eles encontre os publicados, e retorne somente os
↪recentes.
```

Nossos exemplos, foram definidos na classe da própria tabela, porém, você pode ver como um *behavior* o ajudará a automatizar muitos processos e como a reutilização de código é feito no CakePHP leia mais em *Behaviors (Comportamentos)*.

Em uma necessidade de mudar os resultados após uma busca, deve usar a função *Modifying Results with Map/Reduce* para isto. Isto substituí o antigo ‘afterFind’ na versão anterior do Cake. que por sinal trouxe clareza, mais agilidade no processo e menos consumo de memória.

Buscadores dinâmicos

CakePHP’s ORM provê uma dinâmica na construção de métodos de busca, onde na chamada do método poderá apenas adicionar o nome do campo que deseja buscar. Por exemplo, se você quer buscar usuários por seu nome gostará de:

```
// No controller
// Duas chamadas iguais.
$query = $this->Users->findByUsername('joebob');
$query = $this->Users->findAllByUsername('joebob');

// Na tabela
$users = TableRegistry::get('Users');
// Duas chamadas também iguais.
$query = $users->findByUsername('joebob');
$query = $users->findAllByUsername('joebob');
```

Pode usar também múltiplos campos na pesquisa:

```
$query = $users->findAllByUsernameAndApproved('joebob', 1);
//Retorne usuários com Joebob e eles devem estar aprovados ou = 1
```

Use a condição OR expressa:

```
$query = $users->findAllByUsernameOrEmail('joebob', 'joe@example.com');
//Retorne usuário com nome joebob ou que possua o email joe@example.com
```

Neste caso, ao usar ‘OR’ ou ‘AND’ voce não pode combinar os dois em único método. Também não será possível associar dados com o atributo `contain`, pois não é compatível com buscas dinâmicas. Lembre-se dos nossos queridos *Personalizando Métodos de Consulta* eles podem fazer esse trabalho para você com consultas complexas. Por ultimos combine suas buscas personalizadas com as dinâmicas:

```
$query = $users->findTrollsByUsername('bro');
// Procure pelos trolls, esses trolls devem username = bro
```

Abaixo um jeito mais organizado:

```
$users->find('trolls', [
    'conditions' => ['username' => 'bro']
]);
```

Caso tenha objeto Query retornado da busca dinâmica você necessitará de chamar `first()` Se quer o primeiro resultado.

Nota: Esses métodos de busca podem ser simples, porém eles trazem uma sobrecarga adicional, pelo fato de ser necessário entender as expressões.

Retornando Dados Associados

Quando desejar alguns dados associados ou um filtro baseado nesses dados associados, terá dois caminhos para atingir seu objetivo:

- use CakePHP ORM query functions like `contain()` and `matching()`
- use join functions like `innerJoin()`, `leftJoin()`, and `rightJoin()`

Use `contain()` quando desejar carregar uma entidade e seus dados associados. `contain()` aplicará uma condição adicional aos dados relacionados, porém você não poderá aplicar condições nesses dados baseado nos dados relacionais. Mais detalhes veja `contain()` em *Eager Loading Associations*.

`matching()` se você deseja aplicar condições na sua entidade baseado nos dados relacionais, deve usar isto. Por exemplo, você quer carregar todos os artigos que tem uma tag específica neles. Mais detalhes veja `matching()`, em *Filtering by Associated Data*.

Caso prefira usar a função join, veja mais informações em *adding-joins*.

Eager Loading Associations

By default CakePHP does not load **any** associated data when using `find()`. You need to ‘contain’ or eager-load each association you want loaded in your results.

Eager loading helps avoid many of the potential performance problems surrounding lazy-loading in an ORM. The queries generated by eager loading can better leverage joins, allowing more efficient queries to be made. In CakePHP you define eager loaded associations using the ‘contain’ method:

```
// In a controller or table method.

// As an option to find()
$query = $articles->find('all', ['contain' => ['Authors', 'Comments']]);
```

```
// As a method on the query object
$query = $articles->find('all');
$query->contain(['Authors', 'Comments']);
```

The above will load the related author and comments for each article in the result set. You can load nested associations using nested arrays to define the associations to be loaded:

```
$query = $articles->find()->contain([
    'Authors' => ['Addresses'], 'Comments' => ['Authors']
]);
```

Alternatively, you can express nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Authors.Addresses',
    'Comments.Authors'
]);
```

You can eager load associations as deep as you like:

```
$query = $products->find()->contain([
    'Shops.Cities.Countries',
    'Shops.Managers'
]);
```

If you need to reset the containments on a query you can set the second argument to `true`:

```
$query = $articles->find();
$query->contain(['Authors', 'Comments'], true);
```

Passing Conditions to Contain

When using `contain()` you are able to restrict the data returned by the associations and filter them by conditions:

```
// In a controller or table method.

$query = $articles->find()->contain([
    'Comments' => function ($q) {
        return $q
            ->select(['body', 'author_id'])
            ->where(['Comments.approved' => true]);
    }
]);
```

This also works for pagination at the Controller level:

```
$this->paginate['contain'] = [
    'Comments' => function (\Cake\ORM\Query $query) {
        return $query->select(['body', 'author_id'])
            ->where(['Comments.approved' => true]);
    }
];
```

Nota: When you limit the fields that are fetched from an association, you **must** ensure that the foreign key columns are selected. Failing to select foreign key fields will cause associated data to not be present in the final result.

It is also possible to restrict deeply-nested associations using the dot notation:

```
$query = $articles->find()->contain([
    'Comments',
    'Authors.Profiles' => function ($q) {
        return $q->where(['Profiles.is_published' => true]);
    }
]);
```

If you have defined some custom finder methods in your associated table, you can use them inside `contain()`:

```
// Bring all articles, but only bring the comments that are approved and
// popular.
$query = $articles->find()->contain([
    'Comments' => function ($q) {
        return $q->find('approved')->find('popular');
    }
]);
```

Nota: For `BelongsTo` and `HasOne` associations only the `where` and `select` clauses are used when loading the associated records. For the rest of the association types you can use every clause that the query object provides.

If you need full control over the query that is generated, you can tell `contain()` to not append the `foreignKey` constraints to the generated query. In that case you should use an array passing `foreignKey` and `queryBuilder`:

```
$query = $articles->find()->contain([
    'Authors' => [
        'foreignKey' => false,
        'queryBuilder' => function ($q) {
            return $q->where(...); // Full conditions for filtering
        }
    ]
]);
```

If you have limited the fields you are loading with `select()` but also want to load fields off of contained associations, you can pass the association object to `select()`:

```
// Select id & title from articles, but all fields off of Users.
$query = $articles->find()
    ->select(['id', 'title'])
    ->select($articlesTable->Users)
    ->contain(['Users']);
```

Alternatively, if you have multiple associations, you can use `autoFields()`:

```
// Select id & title from articles, but all fields off of Users, Comments
// and Tags.
$query->select(['id', 'title'])
    ->contain(['Comments', 'Tags'])
    ->autoFields(true)
    ->contain(['Users' => function ($q) {
```

```
        return $q->autoFields(true);
    }]);
```

Novo na versão 3.1: Selecting columns via an association object was added in 3.1

Sorting Contained Associations

When loading HasMany and BelongsToMany associations, you can use the `sort` option to sort the data in those associations:

```
$query->contain([
    'Comments' => [
        'sort' => ['Comment.created' => 'DESC']
    ]
]);
```

Filtering by Associated Data

A fairly common query case with associations is finding records ‘matching’ specific associated data. For example if you have ‘Articles belongsToMany Tags’ you will probably want to find Articles that have the CakePHP tag. This is extremely simple to do with the ORM in CakePHP:

```
// In a controller or table method.

$query = $articles->find();
$query->matching('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

You can apply this strategy to HasMany associations as well. For example if ‘Authors HasMany Articles’, you could find all the authors with recently published articles using the following:

```
$query = $authors->find();
$query->matching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new DateTime('-10 days')]);
});
```

Filtering by deep associations is surprisingly easy, and the syntax should be already familiar to you:

```
// In a controller or table method.
$query = $products->find()->matching(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);

// Bring unique articles that were commented by 'markstory' using passed variable
// Dotted matching paths should be used over nested matching() calls
$username = 'markstory';
$query = $articles->find()->matching('Comments.Users', function ($q) use ($username) {
    return $q->where(['username' => $username]);
});
```

Nota: As this function will create an `INNER JOIN`, you might want to consider calling `distinct` on the find query as you might get duplicate rows if your conditions don't exclude them already. This might be the case, for example, when the same users comments more than once on a single article.

The data from the association that is 'matched' will be available on the `_matchingData` property of entities. If you both match and contain the same association, you can expect to get both the `_matchingData` and standard association properties in your results.

Using `innerJoinWith`

Using the `matching()` function, as we saw already, will create an `INNER JOIN` with the specified association and will also load the fields into the result set.

There may be cases where you want to use `matching()` but are not interested in loading the fields into the result set. For this purpose, you can use `innerJoinWith()`:

```
$query = $articles->find();
$query->innerJoinWith('Tags', function ($q) {
    return $q->where(['Tags.name' => 'CakePHP']);
});
```

The `innerJoinWith()` method works the same as `matching()`, that means that you can use dot notation to join deeply nested associations:

```
$query = $products->find()->innerJoinWith(
    'Shops.Cities.Countries', function ($q) {
        return $q->where(['Countries.name' => 'Japan']);
    }
);
```

Again, the only difference is that no additional columns will be added to the result set, and no `_matchingData` property will be set.

Novo na versão 3.1: `Query::innerJoinWith()` was added in 3.1

Using `notMatching`

The opposite of `matching()` is `notMatching()`. This function will change the query so that it filters results that have no relation to the specified association:

```
// In a controller or table method.

$query = $articlesTable
    ->find()
    ->notMatching('Tags', function ($q) {
        return $q->where(['Tags.name' => 'boring']);
    });
```

The above example will find all articles that were not tagged with the word `boring`. You can apply this method to `HasMany` associations as well. You could, for example, find all the authors with no published articles in the last 10 days:

```
$query = $authorsTable
->find()
->notMatching('Articles', function ($q) {
    return $q->where(['Articles.created >=' => new \DateTime('-10 days')]);
});
```

It is also possible to use this method for filtering out records not matching deep associations. For example, you could find articles that have not been commented on by a certain user:

```
$query = $articlesTable
->find()
->notMatching('Comments.Users', function ($q) {
    return $q->where(['username' => 'jose']);
});
```

Since articles with no comments at all also satisfy the condition above, you may want to combine `matching()` and `notMatching()` in the same query. The following example will find articles having at least one comment, but not commented by a certain user:

```
$query = $articlesTable
->find()
->notMatching('Comments.Users', function ($q) {
    return $q->where(['username' => 'jose']);
})
->matching('Comments');
```

Nota: As `notMatching()` will create a `LEFT JOIN`, you might want to consider calling `distinct` on the `find` query as you can get duplicate rows otherwise.

Keep in mind that contrary to the `matching()` function, `notMatching()` will not add any data to the `_matchingData` property in the results.

Novo na versão 3.1: `Query::notMatching()` was added in 3.1

Using `leftJoinWith`

On certain occasions you may want to calculate a result based on an association, without having to load all the records for it. For example, if you wanted to load the total number of comments an article has along with all the article data, you can use the `leftJoinWith()` function:

```
$query = $articlesTable->find();
$query->select(['total_comments' => $query->func()->count('Comments.id')])
->leftJoinWith('Comments')
->group(['Articles.id'])
->autoFields(true);
```

The results for the above query will contain the article data and the `total_comments` property for each of them.

`leftJoinWith()` can also be used with deeply nested associations. This is useful, for example, for bringing the count of articles tagged with a certain word, per author:

```
$query = $authorsTable
->find()
->select(['total_articles' => $query->func()->count('Articles.id')])
->leftJoinWith('Articles.Tags', function ($q) {
```

```

        return $q->where(['Tags.name' => 'awesome']);
    })
    ->group(['Authors.id'])
    ->autoFields(true);

```

This function will not load any columns from the specified associations into the result set.

Novo na versão 3.1: `Query::leftJoinWith()` was added in 3.1

Changing Fetching Strategies

As you may know already, `belongsTo` and `hasOne` associations are loaded using a `JOIN` in the main finder query. While this improves query and fetching speed and allows for creating more expressive conditions when retrieving data, this may be a problem when you want to apply certain clauses to the finder query for the association, such as `order()` or `limit()`.

For example, if you wanted to get the first comment of an article as an association:

```

$articles->hasOne('FirstComment', [
    'className' => 'Comments',
    'foreignKey' => 'article_id'
]);

```

In order to correctly fetch the data from this association, we will need to tell the query to use the `select` strategy, since we want order by a particular column:

```

$query = $articles->find()->contain([
    'FirstComment' => [
        'strategy' => 'select',
        'queryBuilder' => function ($q) {
            return $q->order(['FirstComment.created' => 'ASC'])->limit(1);
        }
    ]
]);

```

Dynamically changing the strategy in this way will only apply to a specific query. If you want to make the strategy change permanent you can do:

```

$articles->FirstComment->strategy('select');

```

Using the `select` strategy is also a great way of making associations with tables in another database, since it would not be possible to fetch records using `joins`.

Fetching With The Subquery Strategy

As your tables grow in size, fetching associations from them can become slower, especially if you are querying big batches at once. A good way of optimizing association loading for `hasMany` and `belongsToMany` associations is by using the subquery strategy:

```

$query = $articles->find()->contain([
    'Comments' => [
        'strategy' => 'subquery',
        'queryBuilder' => function ($q) {
            return $q->where(['Comments.approved' => true]);
        }
    ]
]);

```

```
    ]  
  });
```

The result will remain the same as with using the default strategy, but this can greatly improve the query and fetching time in some databases, in particular it will allow to fetch big chunks of data at the same time in databases that limit the amount of bound parameters per query, such as **Microsoft SQL Server**.

You can also make the strategy permanent for the association by doing:

```
$articles->Comments->strategy('subquery');
```

Lazy Loading Associations

While CakePHP makes it easy to eager load your associations, there may be cases where you need to lazy-load associations. You should refer to the *lazy-load-associations* and *loading-additional-associations* sections for more information.

Working with Result Sets

Once a query is executed with `all()`, you will get an instance of `Cake\ORM\ResultSet`. This object offers powerful ways to manipulate the resulting data from your queries. Like Query objects, ResultSets are a *Collection* and you can use any collection method on ResultSet objects.

Result set objects will lazily load rows from the underlying prepared statement. By default results will be buffered in memory allowing you to iterate a result set multiple times, or cache and iterate the results. If you need work with a data set that does not fit into memory you can disable buffering on the query to stream results:

```
$query->bufferResults(false);
```

Turning buffering off has a few caveats:

1. You will not be able to iterate a result set more than once.
2. You will also not be able to iterate & cache the results.
3. Buffering cannot be disabled for queries that eager load `hasMany` or `belongsToMany` associations, as these association types require eagerly loading all results so that dependent queries can be generated. This limitation is not present when using the `subquery` strategy for those associations.

Aviso: Streaming results will still allocate memory for the entire results when using PostgreSQL and SQL Server. This is due to limitations in PDO.

Result sets allow you to cache/serialize or JSON encode results for API results:

```
// In a controller or table method.  
$results = $query->all();  
  
// Serialized  
$serialized = serialize($results);  
  
// Json  
$json = json_encode($results);
```

Both serializing and JSON encoding result sets work as you would expect. The serialized data can be unserialized into a working result set. Converting to JSON respects hidden & virtual field settings on all entity objects within a result set.

In addition to making serialization easy, result sets are a ‘Collection’ object and support the same methods that *collection objects* do. For example, you can extract a list of unique tags on a collection of articles by running:

```
// In a controller or table method.
$articles = TableRegistry::get('Articles');
$query = $articles->find()->contain(['Tags']);

$reducer = function ($output, $value) {
    if (!in_array($value, $output)) {
        $output[] = $value;
    }
    return $output;
};

$uniqueTags = $query->all()
    ->extract('tags.name')
    ->reduce($reducer, []);
```

Some other examples of the collection methods being used with result sets are:

```
// Filter the rows by a calculated property
$filtered = $results->filter(function ($row) {
    return $row->is_recent;
});

// Create an associative array from result properties
$articles = TableRegistry::get('Articles');
$results = $articles->find()->contain(['Authors'])->all();

$authorList = $results->combine('id', 'author.name');
```

The *Collections (Coleções)* chapter has more detail on what can be done with result sets using the collections features. The *format-results* section show how you can add calculated fields, or replace the result set.

Getting the First & Last Record From a ResultSet

You can use the `first()` and `last()` methods to get the respective records from a result set:

```
$result = $articles->find('all')->all();

// Get the first and/or last result.
$row = $result->first();
$row = $result->last();
```

Getting an Arbitrary Index From a ResultSet

You can use `skip()` and `first()` to get an arbitrary record from a ResultSet:

```
$result = $articles->find('all')->all();
```

```
// Get the 5th record
$row = $result->skip(4)->first();
```

Checking if a Query or ResultSet is Empty

You can use the `isEmpty()` method on a Query or ResultSet object to see if it has any rows in it. Calling `isEmpty()` on a Query object will evaluate the query:

```
// Check a query.
$query->isEmpty();

// Check results
$results = $query->all();
$results->isEmpty();
```

Loading Additional Associations

Once you've created a result set, you may need to load additional associations. This is the perfect time to lazily eager load data. You can load additional associations using `loadInto()`:

```
$articles = $this->Articles->find()->all();
$withMore = $this->Articles->loadInto($articles, ['Comments', 'Users']);
```

You can eager load additional data into a single entity, or a collection of entities.

Modifying Results with Map/Reduce

More often than not, find operations require post-processing the data that is found in the database. While entities' getter methods can take care of most of the virtual property generation or special data formatting, sometimes you need to change the data structure in a more fundamental way.

For those cases, the Query object offers the `mapReduce()` method, which is a way of processing results once they are fetched from the database.

A common example of changing the data structure is grouping results together based on certain conditions. For this task we can use the `mapReduce()` function. We need two callable functions the `$mapper` and the `$reducer`. The `$mapper` callable receives the current result from the database as first argument, the iteration key as second argument and finally it receives an instance of the MapReduce routine it is running:

```
$mapper = function ($article, $key, $mapReduce) {
    $status = 'published';
    if ($article->isDraft() || $article->isInReview()) {
        $status = 'unpublished';
    }
    $mapReduce->emitIntermediate($article, $status);
};
```

In the above example `$mapper` is calculating the status of an article, either published or unpublished, then it calls `emitIntermediate()` on the MapReduce instance. This method stores the article in the list of articles labelled as either published or unpublished.

The next step in the map-reduce process is to consolidate the final results. For each status created in the mapper, the `$reducer` function will be called so you can do any extra processing. This function will receive the list of articles in

a particular “bucket” as the first parameter, the name of the “bucket” it needs to process as the second parameter, and again, as in the `mapper()` function, the instance of the MapReduce routine as the third parameter. In our example, we did not have to do any extra processing, so we just `emit()` the final results:

```
$reducer = function ($articles, $status, $mapReduce) {
    $mapReduce->emit($articles, $status);
};
```

Finally, we can put these two functions together to do the grouping:

```
$articlesByStatus = $articles->find()
    ->where(['author_id' => 1])
    ->mapReduce($mapper, $reducer);

foreach ($articlesByStatus as $status => $articles) {
    echo sprintf("There are %d %s articles", count($articles), $status);
}
```

The above will output the following lines:

```
There are 4 published articles
There are 5 unpublished articles
```

Of course, this is a simplistic example that could actually be solved in another way without the help of a map-reduce process. Now, let’s take a look at another example in which the reducer function will be needed to do something more than just emitting the results.

Calculating the most commonly mentioned words, where the articles contain information about CakePHP, as usual we need a mapper function:

```
$mapper = function ($article, $key, $mapReduce) {
    if (strpos('cakephp', $article['body']) === false) {
        return;
    }

    $words = array_map('strtolower', explode(' ', $article['body']));
    foreach ($words as $word) {
        $mapReduce->emitIntermediate($article['id'], $word);
    }
};
```

It first checks for whether the “cakephp” word is in the article’s body, and then breaks the body into individual words. Each word will create its own bucket where each article id will be stored. Now let’s reduce our results to only extract the count:

```
$reducer = function ($occurrences, $word, $mapReduce) {
    $mapReduce->emit(count($occurrences), $word);
}
```

Finally, we put everything together:

```
$articlesByStatus = $articles->find()
    ->where(['published' => true])
    ->andWhere(['published_date >=' => new DateTime('2014-01-01')])
    ->hydrate(false)
    ->mapReduce($mapper, $reducer);
```

This could return a very large array if we don’t clean stop words, but it could look something like this:

```
[
    'cakephp' => 100,
    'awesome' => 39,
    'impressive' => 57,
    'outstanding' => 10,
    'mind-blowing' => 83
]
```

One last example and you will be a map-reduce expert. Imagine you have a `friends` table and you want to find “fake friends” in our database, or better said, people who do not follow each other. Let’s start with our `mapper()` function:

```
$mapper = function ($rel, $key, $mr) {
    $mr->emitIntermediate($rel['source_user_id'], $rel['target_user_id']);
    $mr->emitIntermediate($rel['target_user_id'], $rel['source_target_id']);
};
```

We just duplicated our data to have a list of users each other user follows. Now it’s time to reduce it. For each call to the reducer, it will receive a list of followers per user:

```
// $friends list will look like
// repeated numbers mean that the relationship existed in both directions
[2, 5, 100, 2, 4]

$reducer = function ($friendsList, $user, $mr) {
    $friends = array_count_values($friendsList);
    foreach ($friends as $friend => $count) {
        if ($count < 2) {
            $mr->emit($friend, $user);
        }
    }
}
```

And we supply our functions to a query:

```
$fakeFriends = $friends->find()
    ->hydrate(false)
    ->mapReduce($mapper, $reducer)
    ->toArray();
```

This would return an array similar to this:

```
[
    1 => [2, 4],
    3 => [6]
    ...
]
```

The resulting array means, for example, that user with id 1 follows users 2 and 4, but those do not follow 1 back.

Stacking Multiple Operations

Using `mapReduce` in a query will not execute it immediately. The operation will be registered to be run as soon as the first result is attempted to be fetched. This allows you to keep chaining additional methods and filters to the query even after adding a map-reduce routine:

```
$query = $articles->find()
    ->where(['published' => true])
    ->mapReduce($mapper, $reducer);

// At a later point in your app:
$query->where(['created >=' => new DateTime('1 day ago')]);
```

This is particularly useful for building custom finder methods as described in the *Personalizando Métodos de Consulta* section:

```
public function findPublished(Query $query, array $options)
{
    return $query->where(['published' => true]);
}

public function findRecent(Query $query, array $options)
{
    return $query->where(['created >=' => new DateTime('1 day ago')]);
}

public function findCommonWords(Query $query, array $options)
{
    // Same as in the common words example in the previous section
    $mapper = ...;
    $reducer = ...;
    return $query->mapReduce($mapper, $reducer);
}

$commonWords = $articles
    ->find('commonWords')
    ->find('published')
    ->find('recent');
```

Moreover, it is also possible to stack more than one mapReduce operation for a single query. For example, if we wanted to have the most commonly used words for articles, but then filter it to only return words that were mentioned more than 20 times across all articles:

```
$mapper = function ($count, $word, $mr) {
    if ($count > 20) {
        $mr->emit($count, $word);
    }
};

$articles->find('commonWords')->mapReduce($mapper);
```

Removing All Stacked Map-reduce Operations

Under some circumstances you may want to modify a Query object so that no mapReduce operations are executed at all. This can be done by calling the method with both parameters as null and the third parameter (overwrite) as true:

```
$query->mapReduce(null, null, true);
```

Validando dados

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹¹⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Validando dados antes de construir entidades

Aplicando regras da aplicação

Usando um Conjunto de Validação Diferente

Usando um Conjunto de Validação Diferente para Associações

Salvando Dados

```
class Cake\ORM\Table
```

Depois que você *carregou seus dados* provavelmente vai querer atualizar e salvar as alterações.

Visão Geral Sobre Salvando Dados

Aplicações geralmente terá algumas maneiras de como os dados são salvos. A primeira é, obviamente, através de formulários web e a outra é por geração direta ou alterando dados no código para enviar ao banco de dados.

Inserindo Dados

A maneira mais fácil de inserir dados no banco de dados é criando uma nova entidade e passando ela pro método `save()` na classe `Table`:

```
use Cake\ORM\TableRegistry;

$articlesTable = TableRegistry::get('Articles');
$article = $articlesTable->newEntity();

$article->title = 'A New Article';
$article->body = 'This is the body of the article';

if ($articlesTable->save($article)) {
    // The $article entity contains the id now
    $id = $article->id;
}
```

Atualizando Dados

Atualizar seus dados é igualmente fácil, e o método `save()` também é usado para esse propósito:

¹¹⁰ <https://github.com/cakephp/docs>

```
use Cake\ORM\TableRegistry;

$articlesTable = TableRegistry::get('Articles');
$article = $articlesTable->get(12); // Return article with id 12

$article->title = 'CakePHP is THE best PHP framework!';
$articlesTable->save($article);
```

CakePHP saberá quando deve realizar uma inserção ou atualização com base no valor de retorno do método `isNew()`. Entidades que foram obtidas com `get()` ou `find()` sempre retornará `false` quando `isNew()` é chamado nelas.

Salvando com Associações

Por padrão o método `save()` também salvará associações de um nível:

```
$articlesTable = TableRegistry::get('Articles');
$author = $articlesTable->Authors->findByUserName('mark')->first();

$article = $articlesTable->newEntity();
$article->title = 'An article by mark';
$article->author = $author;

if ($articlesTable->save($article)) {
    // A chave estrangeira foi atribuída automaticamente
    echo $article->author_id;
}
```

O método `save()` também é capaz de criar novos registros para associações:

```
$firstComment = $articlesTable->Comments->newEntity();
$firstComment->body = 'The CakePHP features are outstanding';

$secondComment = $articlesTable->Comments->newEntity();
$secondComment->body = 'CakePHP performance is terrific!';

$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEntity();
$tag2->name = 'awesome';

$article = $articlesTable->get(12);
$article->comments = [$firstComment, $secondComment];
$article->tags = [$tag1, $tag2];

$articlesTable->save($article);
```

Associe Muitos para Muitos (N para N) registros

O exemplo anterior demonstra como associar algumas tags a um artigo. Outra maneira de realizar a mesma coisa é usando o método `link()` na associação:

```
$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag2 = $articlesTable->Tags->newEntity();
$tag2->name = 'awesome';

$articlesTable->Tags->link($article, [$tag1, $tag2]);
```

Salvando Dados da Tabela de Ligação

Salvar dados na tabela de ligação é realizado usando a propriedade especial `_joinData`. Esta propriedade deve ser um instância de `Entity` da classe `Table` de ligação:

```
// Link records for the first time.
$tag1 = $articlesTable->Tags->findByName('cakephp')->first();
$tag1->_joinData = $articlesTable->ArticlesTags->newEntity();
$tag1->_joinData->tagComment = 'The CakePHP ORM is so powerful!';

$articlesTable->Tags->link($article, [$tag1]);

// Update an existing association.
$article = $articlesTable->get(1, ['contain' => ['Tags']]);
$article->tags[0]->_joinData->tagComment = 'Fresh comment.'

// Necessary because we are changing a property directly
$article->dirty('tags', true);

$articlesTable->save($article, ['associated' => ['Tags']]);
```

Você também pode criar / atualizar informações na tabela de ligação utilizando `newEntity()` ou `patchEntity()`. Os seus dados de POST devem parecer:

```
$data = [
    'title' => 'My great blog post',
    'body' => 'Some content that goes on for a bit.',
    'tags' => [
        [
            'id' => 10,
            '_joinData' => [
                'tagComment' => 'Great article!',
            ]
        ],
    ],
];
$articlesTable->newEntity($data, ['associated' => ['Tags']]);
```

Remover Associação Muitos para Muitos (N para N) Registros

A remoção de associação Muitos para Muitos registros é realizada através do método `unlink()`:

```
$tags = $articlesTable
    ->Tags
    ->find()
    ->where(['name IN' => ['cakephp', 'awesome']])
    ->toArray();

$articlesTable->Tags->unlink($article, $tags);
```

Quando modificando registros, configurando ou alterando diretamente as propriedades, nenhuma validação é realizada, que é um problema quando está aceitando dados de formulário. As seções seguintes demonstrarão como converter eficientemente dados de formulário em entidades que podem ser validadas e salva.

Convertendo Dados de Requisição em Entidades

Antes de editar e salvar os dados de volta no seu banco de dados, você precisará converter os dados da requisição, de array mantido na requisição em entidades que o ORM utiliza. A classe Table fornece uma maneira fácil e eficiente de converter uma ou várias entidades dos dados de requisição. Você pode converter uma entidade usando:

```
//No controller
$articles = TableRegistry::get('Articles');

// Valida e converte em um objeto do tipo Entity
$entity = $articles->newEntity($this->request->getData());
```

Nota: Se você estiver usando newEntity() e as entidades resultantes estão faltando algum ou todos os dados passados, verifique se as colunas que deseja definir estão listadas na propriedade \$_accessible da sua entidade. Consulte [Atribuição em Massa](#).

Os dados da requisição devem seguir a estrutura de suas entidades. Por exemplo, se você tem um artigo, que pertence a um usuário, e tem muitos comentários, os seus dados de requisição devem ser semelhante:

```
$data = [
    'title' => 'CakePHP For the Win',
    'body' => 'Baking with CakePHP makes web development fun!',
    'user_id' => 1,
    'user' => [
        'username' => 'mark'
    ],
    'comments' => [
        ['body' => 'The CakePHP features are outstanding'],
        ['body' => 'CakePHP performance is terrific!'],
    ]
];
```

Por padrão, o método newEntity() valida os dados que são passados para ele, conforme explicado na seção [Validando dados antes de construir entidades](#). Se você deseja pular a validação de dados, informe a opção 'validate' => false:

```
$entity = $articles->newEntity($data, ['validate' => false]);
```

Ao criar formulários que salvam associações aninhadas, você precisa definir quais associações devem ser convertidas:

```
// No controller
$articles = TableRegistry::get('Articles');

// Nova entidade com associações aninhadas
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => ['associated' => ['Users']]
    ]
]);
```

O exemplo acima indica que 'Tags', 'Comments' e 'Users' para os artigos devem ser convertidos. Alternativamente, você pode usar a notação de ponto (dot notation) por brevidade:

```
// No controller
$articles = TableRegistry::get('Articles');
```

```
// Nova entidade com associações aninhada usando notação de ponto
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => ['Tags', 'Comments.Users']
]);
```

Você também pode desativar a conversão de possíveis associações aninhadas como:

```
$entity = $articles->newEntity($data, ['associated' => []]);
// ou...
$entity = $articles->patchEntity($entity, $data, ['associated' => []]);
```

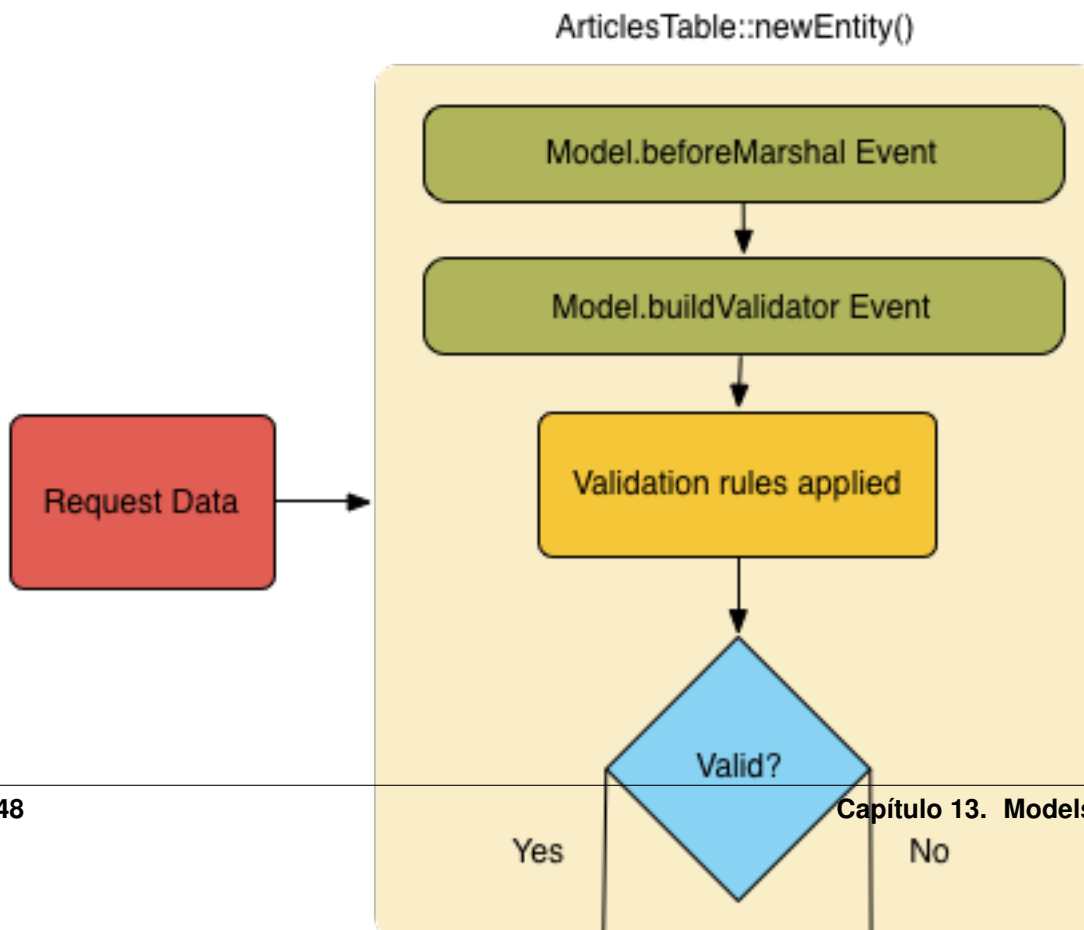
Os dados associados também são validados por padrão, a menos que seja informado o contrário. Você também pode alterar o conjunto de validação a ser usada por associação:

```
// No controller
$articles = TableRegistry::get('Articles');

// Pular validação na associação de Tags e
// Definir 'signup' como método de validação para Comments.Users
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags' => ['validate' => false],
        'Comments.Users' => ['validate' => 'signup']
    ]
]);
```

O capítulo *Usando um Conjunto de Validação Diferente para Associações* possui mais informações sobre como usar diferentes validadores para associações ao transformar em entidades.

O diagrama a seguir fornece uma visão geral do que acontece dentro dos métodos `newEntity()` ou `patchEntity()`:



Você sempre pode contar de obter uma entidade de volta com `newEntity()`. Se a validação falhar, sua entidade

dade
con-
terá
er-
ros, e
quais-
quer
cam-
pos
invá-
lidos
não
serão
preen-
chidos
na en-
tidade
criada.

Convertendo

Dados de Associação BelongsToMany

Se você está salvando associações belongsToMany, você pode tanto usar uma lista de entidades ou uma lista de ids. Ao usar uma lista de dados de entidade, seus dados de requisição devem parecer com:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        ['tag' => 'CakePHP'],
        ['tag' => 'Internet'],
    ]
];
```

O exemplo acima criará 2 novas tags. Se você deseja associar um artigo com tags existentes, você pode usar uma lista de ids. Seus dados de requisição devem parecer com:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
        '_ids' => [1, 2, 3, 4]
    ]
];
```

Se você precisa associar a alguns belongsToMany registros existentes, e criar novos ao mesmo tempo, você pode usar um formato expandido:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'user_id' => 1,
    'tags' => [
```

```
        ['name' => 'A new tag'],
        ['name' => 'Another new tag'],
        ['id' => 5],
        ['id' => 21]
    ]
};
```

Quando os dados acima são convertidos em entidades, você terá 4 tags. As duas primeiras serão objetos novos, e as outras duas serão referências a registros existentes.

Ao converter dados belongsToMany, você pode desativar a criação de nova entidade, usando a opção `onlyIds`. Quando habilitado, esta opção restringe transformação de belongsToMany para apenas usar a chave `_ids` e ignorar todos os outros dados.

Novo na versão 3.1.0: A opção `onlyIds` foi adicionada na versão 3.1.0

Convertendo Dados de Associação HasMany

Se você deseja atualizar as associações hasMany existentes e atualizar suas propriedades, primeiro você deve garantir que sua entidade seja carregada com a associação hasMany. Você pode então usar dados de requisição semelhantes a:

```
$data = [
    'title' => 'My Title',
    'body' => 'The text',
    'comments' => [
        ['id' => 1, 'comment' => 'Update the first comment'],
        ['id' => 2, 'comment' => 'Update the second comment'],
        ['comment' => 'Create a new comment'],
    ]
];
```

Se você está salvando associações hasMany e deseja vincular a registros existentes, você pode usar o formato `_ids`:

```
$data = [
    'title' => 'My new article',
    'body' => 'The text',
    'user_id' => 1,
    'comments' => [
        '_ids' => [1, 2, 3, 4]
    ]
];
```

Ao converter dados hasMany, você pode desativar a criação de nova entidade, usando a opção `onlyIds`. Quando ativada, esta opção restringe transformação de hasMany para apenas usar a chave `__ids` e ignorar todos os outros dados.

Novo na versão 3.1.0: A opção `onlyIds` foi adicionada na versão 3.1.0

Converendo Vários Registros

Ao criar formulários que cria/atualiza vários registros ao mesmo tempo, você pode usar o método `newEntities()`:

```
// No controller.
$articles = TableRegistry::get('Articles');
$entities = $articles->newEntities($this->request->getData());
```

Nessa situação, os dados de requisição para vários artigos devem parecer com:

```
$data = [
    [
        'title' => 'First post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];
```

Uma vez que você converteu os dados de requisição em entidades, você pode salvar com `save()` e remover com `delete()`:

```
// No controller.
foreach ($entities as $entity) {
    // Salva a entidade
    $articles->save($entity);

    // Remover a entidade
    $articles->delete($entity);
}
```

O exemplo acima executará uma transação separada para cada entidade salva. Se você deseja processar todas as entidades como uma única transação, você pode usar `transactional()`:

```
// No controller.
$articles->getConnection()->transactional(function () use ($articles, $entities) {
    foreach ($entities as $entity) {
        $articles->save($entity, ['atomic' => false]);
    }
});
```

Alterando Campos Acessíveis

Também é possível permitir `newEntity()` escrever em campos não acessíveis. Por exemplo, `id` geralmente está ausente da propriedade `_accessible`. Nesse caso, você pode usar a opção `accessibleFields`. Isso pode ser útil para manter ids de entidades associadas:

```
// No controller
$articles = TableRegistry::get('Articles');
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags', 'Comments' => [
            'associated' => [
                'Users' => [
                    'accessibleFields' => ['id' => true]
                ]
            ]
        ]
    ]
]);
```

O exemplo acima manterá a associação inalterada entre `Comments` e `Users` para a entidade envolvida.

Nota: Se você estiver usando `newEntity()` e as entidades resultantes estão faltando algum ou todos os dados passados, verifique se as colunas que deseja definir estão listadas na propriedade `$_accessible` da sua entidade. Consulte [Atribuição em Massa](#).

Mesclando Dados de Requisição em Entidades

Para atualizar as entidades, você pode escolher de aplicar dados de requisição diretamente em uma entidade existente. Isto tem a vantagem que apenas os campos que realmente mudaram serão salvos, em oposição ao envio de todos os campos para o banco de dados pra ser persistido. Você pode mesclar um array de dados bruto em uma entidade existente usando o método `patchEntity()`:

```
// No controller.
$articles = TableRegistry::get('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $this->request->getData());
$articles->save($article);
```

Validação e patchEntity

Semelhante ao `newEntity()`, o método `patchEntity` validará os dados antes de ser copiado para entidade. O mecanismo é explicado na seção [Validando dados antes de construir entidades](#). Se você deseja desativar a validação, informe a opção `validate` assim:

```
// No controller.
$articles = TableRegistry::get('Articles');
$article = $articles->get(1);
$articles->patchEntity($article, $data, ['validate' => false]);
```

Você também pode alterar a regra de validação utilizada pela entidade ou qualquer uma das associações:

```
$articles->patchEntity($article, $this->request->getData(), [
    'validate' => 'custom',
    'associated' => ['Tags', 'Comments.Users' => ['validate' => 'signup']]
]);
```

Patching HasMany and BelongsToMany

Como explicado na seção anterior, os dados de requisição deve seguir a estrutura de sua entidade. O método `patchEntity()` é igualmente capaz de mesclar associações, por padrão, apenas o primeiro nível de associações são mesclados, mas se você deseja controlar a lista de associações a serem mescladas ou mesclar em níveis mais profundos, você pode usar o terceiro parâmetro do método:

```
// No controller.
$associated = ['Tags', 'Comments.Users'];
$article = $articles->get(1, ['contain' => $associated]);
$articles->patchEntity($article, $this->request->getData(), [
    'associated' => $associated
]);
$articles->save($article);
```

As associações são mescladas ao combinar o campo da chave primária nas entidades de origem com os campos correspondentes no array de dados. As associações irão construir novas entidades se nenhuma entidade anterior for encontrada para a propriedade alvo da associação.

Por exemplo, forneça alguns dados de requisição como este:

```
$data = [
    'title' => 'My title',
    'user' => [
        'username' => 'mark'
    ]
];
```

Tentando popular uma entidade sem uma entidade na propriedade user criará uma nova entidade do tipo user:

```
// In a controller.
$entity = $articles->patchEntity(new Article, $data);
echo $entity->user->username; // Echoes 'mark'
```

O mesmo pode ser dito sobre associações hasMany e belongsToMany, com uma advertência importante:

Nota: Para as associações belongsToMany, garanta que a entidade relevante tenha uma propriedade acessível para a entidade associada.

Se um Produto pertence a várias (belongsToMany) Tag:

```
// Na classe da entidade Product
protected $_accessible = [
    // .. outras propriedades
    'tags' => true,
];
```

Nota: Para as associações hasMany e belongsToMany, se houvesse algumas entidades que não pudessem ser correspondidas por chave primaria a um registro no array de dados, então esses registros serão descartados da entidade resultante.

Lembre-se que usando `patchEntity()` ou `patchEntities()` não persiste os dados, isso apenas edita (ou cria) as entidades informadas. Para salvar a entidade você terá que chamar o método `save()` da model Table.

Por exemplo, considere o seguinte caso:

```
$data = [
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'First comment', 'id' => 1],
        ['body' => 'Second comment', 'id' => 2],
    ]
];
$entity = $articles->newEntity($data);
$articles->save($entity);

$newData = [
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
    ]
];
```

```
        ['body' => 'A new comment'],
    ]
];
$articles->patchEntity($entity, $newData);
$articles->save($entity);
```

No final, se a entidade for convertida de volta para um array, você obterá o seguinte resultado:

```
[
    'title' => 'My title',
    'body' => 'The text',
    'comments' => [
        ['body' => 'Changed comment', 'id' => 1],
        ['body' => 'A new comment'],
    ]
];
```

Como você pode ver, o comentário com id 2 não está mais lá, já que ele não pode ser correspondido a nada no array `$newData`. Isso acontece porque CakePHP está refletindo o novo estado descrito nos dados de requisição.

Algumas vantagens adicionais desta abordagem é que isto reduz o número de operações a serem executadas ao persistir a entidade novamente.

Por favor, observe que isso não significa que o comentário com id 2 foi removido do bando de dados, se você deseja remover os comentários para este artigo que não estão presentes na entidade, você pode coletar as chaves primárias e executar uma exclusão de lote para esses que não estão na lista:

```
// Num controller.
$comments = TableRegistry::get('Comments');
$present = (new Collection($entity->comments))->extract('id')->filter()->toArray();
$comments->deleteAll([
    'article_id' => $article->id,
    'id NOT IN' => $present
]);
```

Como você pode ver, isso também ajuda ao criar soluções onde uma associação precisa de ser implementada como um único conjunto.

Você também pode popular várias entidades ao mesmo tempo. As considerações feitas para popular (patch) associações `hasMany` e `belongsToMany` se aplicam para popular várias entidades: As comparação são feitas pelo valor do campo da chave primária e as correspondências que faltam no array das entidades originais serão removidas e não estarão presentes no resultado:

```
// Num controller.
$articles = TableRegistry::get('Articles');
$list = $articles->find('popular')->toArray();
$patched = $articles->patchEntities($list, $this->request->getData());
foreach ($patched as $entity) {
    $articles->save($entity);
}
```

Semelhante de usar `patchEntity()`, você pode usar o terceiro argumento para controlar as associações que serão mescladas em cada uma das entidades no array:

```
// Num controller.
$patched = $articles->patchEntities(
    $list,
    $this->request->getData(),
```

```
['associated' => ['Tags', 'Comments.Users']]
);
```

Modificando Dados de Requisição Antes de Contruir Entidades

Se você precisa modificar dados de requisição antes de converter em entidades, você pode usar o evento `Model.beforeMarshal`. Esse evento deixa você manipular o dados de requisição antes das entidades serem criadas:

```
// Inclua as instruções na área superior do seu arquivo.
use Cake\Event\Event;
use ArrayObject;

// Na classe da sua table ou behavior
public function beforeMarshal(Event $event, ArrayObject $data, ArrayObject $options)
{
    if (isset($data['username'])) {
        $data['username'] = mb_strtolower($data['username']);
    }
}
```

O parâmetro `$data` é uma instância de `ArrayObject`, então você não precisa retornar ele para alterar os dados usado para criar entidades.

O propósito principal do `beforeMarshal` é auxiliar os usuários a passar o processo de validação quando erros simples podem ser automaticamente resolvidos, ou quando os dados precisam ser reestruturados para que ele possa ser colocado nos campos corretos.

O evento `Model.beforeMarshal` é disparado apenas no início do processo de validação, uma das razões é que o `beforeMarshal` é permitido de alterar as regras de validação e opções de salvamento, como o campo `whitelist`. Validação é disparada logo após este evento ser finalizado. Um exemplo comum de alteração de dados antes de ser validado, é retirar espaço no início e final (trimming) de todos os campos antes de salvar:

```
// Inclua as instruções na área superior do seu arquivo.
use Cake\Event\Event;
use ArrayObject;

// Na classe da sua table ou behavior
public function beforeMarshal(Event $event, ArrayObject $data, ArrayObject $options)
{
    foreach ($data as $key => $value) {
        if (is_string($value)) {
            $data[$key] = trim($value);
        }
    }
}
```

Por causa de como o processo de marshalling trabalha, se um campo não passar na validação ele será automaticamente removido do array de dados e não será copiado na entidade. Isso previne que dados inconsistentes entrem no objeto de entidade.

Além disso, os dados em `beforeMarshal` são uma cópia dos dados passados. Isto é assim porque é importante preservar a entrada original do usuário, pois ele pode ser usado em outro lugar.

Validando Dados Antes de Construir Entidades

O capítulo *Validando dados* contém mais informações de como usar os recursos de validação do CakePHP para garantir que os seus dados permaneçam corretos e consistentes.

Evitando Ataques de Atribuição em Massa de Propriedade

Ao criar ou mesclar entidades a partir de dados de requisição, você precisa ser cuidadoso com o que você permite seus usuários de alterar ou incluir nas entidades. Por exemplo, ao enviar um array na requisição contendo o `user_id` um invasor pode alterar o proprietário de um artigo, causando efeitos indesejáveis:

```
// Contém ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->getData();
$entity = $this->patchEntity($entity, $data);
$this->save($entity);
```

Há dois modos de proteger você contra este problema. O primeiro é configurando as colunas padrão que podem ser definidas com segurança a partir de um requisição usando o recurso *Atribuição em Massa* nas entidades.

O segundo modo é usando a opção `fieldList` ao criar ou mesclar dados em uma entidade:

```
// Contem ['user_id' => 100, 'title' => 'Hacked!'];
$data = $this->request->getData();

// Apenas permite alterar o campo title
$entity = $this->patchEntity($entity, $data, [
    'fieldList' => ['title']
]);
$this->save($entity);
```

Você também pode controlar quais propriedades poder ser atribuídas para associações:

```
// Apenas permite alterar o title e tags
// e nome da tag é a única coluna que pode ser definido
$entity = $this->patchEntity($entity, $data, [
    'fieldList' => ['title', 'tags'],
    'associated' => ['Tags' => ['fieldList' => ['name']]]
]);
$this->save($entity);
```

Usar este recurso é útil quando você tem várias funções diferentes que seus usuários podem acessar, e você deseja que eles editem difentes dados baseados em seus privilégios.

A opção `fieldList` também é aceita nos métodos `newEntity()`, `newEntities()` e `patchEntities()`.

Salvando Entidades

```
Cake\ORM\Table::save(Entity $entity, array $options = [])
```

Ao salvar dados de requisição no seu banco de dados, você primeiro precisa hidratar (`hydrate`) uma nova entidade usando `newEntity()` para passar no `save()`. Por exemplo:

```
// Num controller
$articles = TableRegistry::get('Articles');
$article = $articles->newEntity($this->request->getData());
if ($articles->save($article)) {
```



```
// ...
}
```

O ORM usa o método `isNew()` em uma entidade para determinar quando um insert ou update deve ser realizado ou não. Se o método `isNew()` retorna `true` e a entidade tiver um valor de chave primária, então será emitida uma query `'exists'`. A query `'exists'` pode ser suprimida informando a opção `'checkExisting' => false` no argumento `$options`:

```
$articles->save($article, ['checkExisting' => false]);
```

Uma vez, que você carregou algumas entidades, você provavelmente desejará modificar elas e atualizar em seu banco de dados. Este é um exercício bem simples no CakePHP:

```
$articles = TableRegistry::get('Articles');
$article = $articles->find('all')->where(['id' => 2])->first();

$article->title = 'My new title';
$articles->save($article);
```

Ao salvar, CakePHP irá *aplicar suas regras*, e envolver a operação de salvar em uma transação de banco de dados. Também atualizará as propriedades que mudaram. A chamada `save()` do exemplo acima geraria SQL como:

```
UPDATE articles SET title = 'My new title' WHERE id = 2;
```

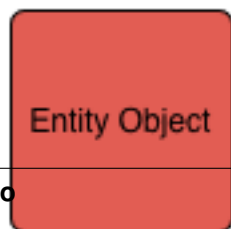
Se você tem uma nova entidade, o seguinte SQL seria gerado:

```
INSERT INTO articles (title) VALUES ('My new title');
```

Quando uma entidade é salva algumas coisas acontecem:

1. A verificação de regras será iniciada se não estiver desativada.
2. A verificação de regras irá disparar o evento `Model.beforeRules`. Se esse evento for parado, a operação de salvamento falhará e retornará `false`.
3. As regras serão verificadas. Se a entidade está sendo criada, as regras `create` serão usadas. Se a entidade estiver sendo atualizada, as regras `update` serão usadas.
4. O evento `Model.afterRules` será disparado.
5. O evento `Model.beforeSave` será disparado. Se ele for parado, o processo de salvamento será abortado, e `save()` retornará `false`.
6. As associações de país são salvas. Por exemplo, qualquer associação `belongsTo` listada serão salvas.
7. Os campos modificados na entidade serão salvos.
8. As associações filhas são salvas. Por exemplo, qualquer associação `hasMany`, `hasOne`, ou `belongsToMany` listada serão salvas.
9. O evento `Model.afterSave` será disparado.
10. O evento `Model.afterSaveCommit` será disparado.

O seguinte diagrama ilustra o processo acima:



Consulte a seção *Aplicando regras da aplicação*

para mais
informação
sobre como
criar e usar
regras.

Aviso:

Se nenhuma
alteração
é feita na
entidade
quando
ela é
salva, os
callbacks
não serão
disparado
porque o
salvar não
é execu-
tado.

O método
`save()`
retornará
a entidade
modificada
quando
sucesso,
e `false`
quando fa-
lhar. Você
pode desa-
tivar regras
e/ou transa-
ções usando
o argumento
`$options`
para salvar:

Salvando Associações

Quando você está salvando uma entidade, você também pode escolher de salvar alguma ou todas as entidades associadas. Por padrão, todas as entidades de primeiro nível serão salvas. Por exemplo salvando um Artigo, você também atualizará todas as entidades modificadas (dirty) que são diretamente realiccionadas a tabela de artigos.

Você pode ajustar as associações que são salvas usando a opção

associated:

```
// Num controller.

// Apenas salva a associação de comentários
$articles->save($entity, ['associated' => ['Comments']]);
```

Você pode definir para salvar associações distantes ou profundamente aninhadas usando a notação de pontos (dot notation):

```
// Salva a company (empresa), employees (funcionários) e os addresses (endereço) relacionado
```

```
para cada um deles
$companies->save($entity, ['associated' => ['Employees.Addresses']]);
```

Além disso, você pode combinar a notação de pontos (dot notation) para associações com o array de opções:

```
$companies->save($entity, [
    'associated' => [
        'Employees',
        'Employees.Addresses'
    ]
]);
```

As suas entidades devem ser estruturadas na mesma maneira como elas são quando carregadas do banco de dados. Consulte a documentação do form helper para saber *como criar inputs para associações*.

Se você está construindo ou modificando dados de associação após a construção de suas entidades, você terá que marcar a propriedade da associação como modificado com o método `dirty()`:

```
$company->author->name = 'Master Chef';
$company->dirty('author', true);
```

Salvando Associações BelongsTo

Ao salvar associações `belongsTo`, o ORM espera uma única entidade aninhada nomeada com a singular, *underscored* versão do nome da associação. Por exemplo:

```
// Num controller.
$data = [
    'title' => 'First Post',
    'user' => [
        'id' => 1,
        'username' => 'mark'
    ]
];
$articles = TableRegistry::get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Users']
]);

$articles->save($article);
```

Salvando Associações HasOne

Ao salvar associações `hasOne`, o ORM espera uma única entidade aninhada nomeada com a singular, *underscored* versão do nome da associação. Por exemplo:

```
// Num controller.
$data = [
    'id' => 1,
    'username' => 'cakephp',
    'profile' => [
        'twitter' => '@cakephp'
    ]
];
```

```
$users = TableRegistry::get('Users');
$user = $users->newEntity($data, [
    'associated' => ['Profiles']
]);
$users->save($user);
```

Salvando Associações HasMany

Ao salvar associações hasMany, o ORM espera um array de entidades nomeada com a plural, *underscored* versão do nome da associação. Por exemplo:

```
// Num controller.
$data = [
    'title' => 'First Post',
    'comments' => [
        ['body' => 'Best post ever'],
        ['body' => 'I really like this.']
    ]
];
$articles = TableRegistry::get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Comments']
]);
$articles->save($article);
```

Ao salvar associações hasMany, registros associados serão atualizados ou inseridos. Para os caso em que o registro já tem registros associados no banco de dados, você tem que escolher entre duas estratégias de salvamento:

append Os registros associados são atualizados no banco de dados ou, se não encontrado nenhum registro existente ele é inserido.

replace Todos os registros existentes que não estão presentes nos registros fornecidos serão removidos do banco dados. Apenas os registros fornecidos permanecerão (ou serão inseridos).

Por padrão é utilizado a estratégia de salvamento append. Consulte [Associações HasMany](#) para mais detalhes sobre como definir saveStrategy.

Sempre que você adiciona novos registros a uma associação existente, você sempre deve marcar a propriedade de associação como 'dirty'. Isso permite que o ORM saiba que a propriedade de associação tem que ser persistida:

```
$article->comments[] = $comment;
$article->dirty('comments', true);
```

Sem a chamada ao método dirty() os comentários atualizados não serão salvos.

Salvando Associações BelongsToMany

Ao salvar associações belongsToMany, o ORM espera um array de entidades nomeada com a plural, *underscored* versão do nome da associação. Por exemplo:

```
// Num controller.
$data = [
    'title' => 'First Post',
    'tags' => [
        ['tag' => 'CakePHP'],
    ]
];
```

```
        ['tag' => 'Framework']
    ]
};
$articles = TableRegistry::get('Articles');
$article = $articles->newEntity($data, [
    'associated' => ['Tags']
]);
$articles->save($article);
```

Ao converter dados de requisição em entidades, os métodos `newEntity()` e `newEntities()` processarão ambos, arrays de propriedades, bem como uma lista de ids na chave `_ids`. Utilizando a chave `_ids` facilita a criação de uma caixa de seleção ou checkbox para associações pertence a muitos (belongs to many). Consulte a seção [Convertendo Dados de Requisição em Entidades](#) para mais informações.

Ao salvar associações `belongsToMany`, você tem que escolher entre duas estratégias de salvamento:

append Apenas novos links serão criados entre cada lado dessa associação. Essa estratégia não destruirá links existentes, mesmo se não estiver presente no array de entidades a serem salvas.

replace Ao salvar, os links existentes serão removidos e novos links serão criados na tabela de ligação. Se houver link existente no banco de dados para algumas das entidades a serem salvas, esses links serão atualizados, e não excluídos para então serem salvos novamente.

Consulte [Associações BelongsToMany](#) para detalhes de como definir `saveStrategy`.

Por padrão é utilizado a estratégia `replace`. Sempre que você adiciona novos registros a uma associação existente, você sempre deve marcar a propriedade de associação como `'dirty'`. Isso permite que o ORM saiba que a propriedade de associação tem que ser persistida:

```
$article->tags[] = $tag;
$article->dirty('tags', true);
```

Sem a chamada ao método `dirty()` as tags atualizadas não serão salvas.

Frequentemente você se encontrará querendo fazer uma associação entre duas entidades existentes, por exemplo. Um usuário que é autor de um artigo. Isso é feito usando o método `link()`, como isso:

```
$article = $this->Articles->get($articleId);
$user = $this->Users->get($userId);

$this->Articles->Users->link($article, [$user]);
```

Ao salvar associações `belongsToMany`, pode ser relevante de salvar algumas informações adicionais na tabela de ligação. No exemplo anterior de tags, poderia ser o `vote_type` da pessoa que votou nesse artigo. O `vote_type` pode ser `upvote` ou `downvote` e ele é representado por uma string. A relação é entre `Users` e `Articles`.

Salvando essa associação, e o `vote_type` é feito primeiramente adicionando alguns dados em `_joinData` e então salvando a associação com `link()`, exemplo:

```
$article = $this->Articles->get($articleId);
$user = $this->Users->get($userId);

$user->_joinData = new Entity(['vote_type' => $voteType], ['markNew' => true]);
$this->Articles->Users->link($article, [$user]);
```

Salvando Dados Adicionais na Tabela de Ligação

Em algumas situações a tabela ligando sua associação `BelongsToMany`, terá colunas adicionais nela. CakePHP torna simples salvar propriedade nessas colunas. Cada entidade em uma associação `belongsToMany` tem uma propriedade `__joinData` que contém as colunas adicionais na tabela de ligação. Esses dados podem ser um array ou uma instância de Entity. Por exemplo se `Students BelongsToMany Courses`, nós poderíamos ter uma tabela de ligação que parece com:

```
id | student_id | course_id | days_attended | grade
```

Ao salvar dados, você pode popular as colunas adicionais na tabela de ligação definindo dados na propriedade `__joinData`:

```
$student->courses[0]->__joinData->grade = 80.12;
$student->courses[0]->__joinData->days_attended = 30;

$studentsTable->save($student);
```

A propriedade `__joinData` pode ser uma entity, ou um array de dados, se você estiver salvando entidades construídas a partir de dados de requisição. Ao salvar os dados de tabela de ligação a partir de dados de requisição, seus dados POST devem parecer com:

```
$data = [
    'first_name' => 'Sally',
    'last_name' => 'Parker',
    'courses' => [
        [
            'id' => 10,
            '__joinData' => [
                'grade' => 80.12,
                'days_attended' => 30
            ]
        ],
        // Other courses.
    ]
];
$student = $this->Students->newEntity($data, [
    'associated' => ['Courses.__joinData']
]);
```

Consulte a documentação *Criando Inputs para Dados Associados* para saber como criar inputs com `FormHelper` corretamente.

Salvando Tipos Complexos (Complex Types)

As tabelas são capazes de armazenar dados representados em tipos básicos, como strings, inteiros, flutuante, booleanos, etc. Mas também pode ser estendido para aceitar tipos mais complexos, como arrays ou objects e serializar esses dados em tipos mais simples que podem ser salvos em banco de dados.

Essa funcionalidade é alcançada usando o sistema de tipos personalizados (custom types system). Consulte a seção *Adicionando Tipos Personalizados* para descobrir como criar tipo de coluna personalizada (custom column Types):

```
// No config/bootstrap.php
use Cake\Database\Type;
Type::map('json', 'Cake\Database\Type\JsonType');
```

```
// No src/Model/Table/UsersTable.php
use Cake\Database\Schema\TableSchema;

class UsersTable extends Table
{
    protected function _initializeSchema(TableSchema $schema)
    {
        $schema->columnType('preferences', 'json');
        return $schema;
    }
}
```

O código acima mapeia a coluna `preferences` para o tipo personalizado (custom type) `json`. Isso significa que, ao obter dados dessa coluna, ele será desserializado de uma string JSON no banco de dados e colocado em uma entidade como um array.

Da mesma forma, quando salvo, o array será transformado novamente em sua representação de JSON:

```
$user = new User([
    'preferences' => [
        'sports' => ['football', 'baseball'],
        'books' => ['Mastering PHP', 'Hamlet']
    ]
]);
$usersTable->save($user);
```

Ao usar tipos complexos, é importante validar que os dados que você está recebendo do usuário final são do tipo correto. A falha ao gerir corretamente dados complexos, pode resultar em usuário mal-intencionados serem capazes de armazenar dados que eles normalmente não seriam capaz.

Strict Saving

`Cake\ORM\Table::saveOrFail($entity, $options = [])`

Usar este método lançará uma `Cake\ORM\Exception\PersistenceFailedException` se:

- as verificações das regras de validação falharam
- a entidade contém erros
- o save foi abortado por um callback.

Usar isso pode ser útil quando você estiver realizando operações complexas no banco de dado sem monitoramento humano, por exemplo, dentro de uma tarefa de Shell.

Nota: Se você usar esse método em um controller, certifique-se de tratar a `PersistenceFailedException` que pode ser lançada.

Se você quiser rastrear a entidade que falhou ao salvar, você pode usar o método `Cake\ORM\Exception\PersistenceFailedException::getEntity()`:

```
try {
    $table->saveOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
```



```

    echo $e->getEntity();
}

```

Como isso executa internamente uma chamada ao `Cake\ORM\Table::save()`, todos eventos de save correspondentes serão disparados.

Novo na versão 3.4.1.

Salvando Várias Entidades

`Cake\ORM\Table::saveMany($entities, $options = [])`

Usando esse método você pode salvar várias entidades atômicamente. `$entities` podem ser um array de entidades criadas usando `newEntities()` / `patchEntities()`. `$options` pode ter as mesmas opções aceitas por `save()`:

```

$data = [
    [
        'title' => 'First post',
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];
$articles = TableRegistry::get('Articles');
$entities = $articles->newEntities($data);
$result = $articles->saveMany($entities);

```

O resultado será as entidades atualizadas em caso de sucesso ou `false` em caso de falha.

Novo na versão 3.2.8.

Atualização em Massa

`Cake\ORM\Table::updateAll($fields, $conditions)`

Pode haver momentos em que atualizar linhas individualmente não é eficiente ou necessária. Nesses casos, é mais eficiente usar uma atualização em massa para modificar várias linhas de uma vez só:

```

// Publique todos artigos não publicados
function publishAllUnpublished()
{
    $this->updateAll(
        ['published' => true], // fields
        ['published' => false]); // conditions
}

```

Se você precisa de atualização em massa e usar expressões SQL, você precisará usar um objeto de expressão como `updateAll()` usa prepared statements por baixo dos panos:

```

use Cake\Database\Expression\QueryExpression;

...

function incrementCounters()

```

```
{
    $expression = new QueryExpression('view_count = view_count + 1');
    $this->updateAll([$expression], ['published' => true]);
}
```

Uma atualização em massa será considerada bem-sucedida se uma ou mais linhas forem atualizadas.

Aviso: `updateAll` não irá disparar os eventos `beforeSave`/`afterSave`. Se você precisa deles, primeiro carregue uma coleção de registros e então atualize eles.

`updateAll()` é apenas por conveniência. Você também pode usar essa interface mais flexível:

```
// Publique todos artigos não publicados.
function publishAllUnpublished()
{
    $this->query()
        ->update()
        ->set(['published' => true])
        ->where(['published' => false])
        ->execute();
}
```

Consulte também: *Atualizando Dados*.

Excluindo Dados

```
class Cake\ORM\Table
```

```
Cake\ORM\Table::delete(Entity $entity, $options = [])
```

Depois que você carregou uma entidade, você pode excluir ela chamando o método `delete` da tabela de origem:

```
// Num a controller.
$entity = $this->Articles->get(2);
$result = $this->Articles->delete($entity);
```

Ao excluir entidades algumas coisas acontecem:

1. As *delete rules* serão aplicadas. Se as regras falharem, a exclusão será impedida.
2. O evento `Model.beforeDelete` é disparado. Se esse evento for interrompido, a exclusão será cancelada e o resultado do evento será retornado.
3. A entidade será excluída.
4. Todas as associações dependentes serão excluídas. Se as associações estão sendo excluídas como entidades, eventos adicionais serão disparados.
5. Qualquer registro da tabela de ligação para associação `BelongsToMany` serão removidos.
6. O evento `Model.afterDelete` será disparado.

Por padrão, todas as exclusões acontecem dentro de uma transação. Você pode desativar a transação com a opção `atomic`:

```
$result = $this->Articles->delete($entity, ['atomic' => false]);
```

Exclusão em Cascata

Ao excluir entidades, os dados associados também podem ser excluídos. Se suas associações HasOne e HasMany estão configurados como `dependent`, as operações de exclusão serão ‘cascade’ para essas entidades também. Por padrão entidades em tabelas associadas são removidas usando `Cake\ORM\Table::deleteAll()`. Você pode optar que o ORM carregue as entidades relacionadas, para então excluir individualmente, definindo a opção `cascadeCallbacks` como `true`:

```
// No método initialize de alguma modelo Table
$this->hasMany('Comments', [
    'dependent' => true,
    'cascadeCallbacks' => true,
]);
```

Nota: Configurando `cascadeCallbacks` para `true`, resulta em exclusões consideravelmente mais lentos quando comparado com exclusão em massa. A opção `cascadeCallbacks` apenas deve ser ativada quando sua aplicação tem trabalho importante manipulado por event listeners.

Exclusão em Massa

`Cake\ORM\Table::deleteAll($conditions)`

Pode ter momentos em que excluir linhas individualmente não é eficiente ou útil. Nesses casos, é mais eficiente usar uma exclusão em massa para remover várias linhas de uma vez só:

```
// Exclui todos oss spam
function destroySpam()
{
    return $this->deleteAll(['is_spam' => true]);
}
```

Uma exclusão em massa será considerada bem-sucedida se uma ou mais linhas forem excluídas.

Aviso: `deleteAll` não dispara os eventos `beforeDelete`/`afterDelete`. Se você precisa deles, você precisa, primeiro carregar uma coleção de registros e então excluí-las.

Exclusões Estrita

`Cake\ORM\Table::deleteOrFail($entity, $options = [])`

Usar esse método lançará uma `Cake\ORM\Exception\PersistenceFailedException` se:

- a entidade é nova
- a entidade não tem valor de chave primária
- as verificações das regras da aplicação falharam
- a exclusão foi interrompida por um callback.

Se você deseja rastrear a entidade que falhou ao salvar, você pode usar o método `Cake\ORM\Exception\PersistenceFailedException::getEntity()`:

```
try {
    $table->deleteOrFail($entity);
} catch (\Cake\ORM\Exception\PersistenceFailedException $e) {
    echo $e->getEntity();
}
```

Como isso executa internamente uma chamada ao `Cake\ORM\Table::delete()`, todos eventos de exclusão correspondentes serão disparados.

Novo na versão 3.4.1.

Associações - Conectando tabelas

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹¹¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Associações HasOne

Associações HasMany

Associações BelongsToMany

Behaviors (Comportamentos)

Os behaviors são um modo de organizar e habilitar o reuso de lógica da camada do Model (Modelo). Conceitualmente, eles são semelhantes a traits. No entanto, os behaviors são implementados como classes separadas. Isso permite que eles se conectem aos callbacks de ciclo de vida que os modelos emitem, ao mesmo tempo que fornecem recursos semelhantes a traits.

Os behaviors fornecem uma maneira conveniente de compor comportamentos que são comuns em vários modelos. Por exemplo, CakePHP inclui um `TimestampBehavior`. Vários modelos irão querer campos de timestamp, e a lógica para gerenciar esses campos não é específica para nenhum modelo. São esses tipos de cenários em que os behaviors são perfeitos.

Usando Behaviors

Behaviors fornecem uma maneira fácil de criar partes de lógica horizontalmente reutilizáveis relacionadas às classes de tabela. Você pode estar se perguntando por que os behaviors são classes regulares e não traits. O principal motivo para isso é event listeners. Enquanto as traits permitiriam partes reutilizáveis de lógica, eles complicariam o uso de eventos.

Para adicionar um behavior à sua tabela, você pode chamar o método `addBehavior()`. Geralmente o melhor lugar para fazer isso é no método `initialize()`:

¹¹¹ <https://github.com/cakephp/docs>

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }
}
```

Como acontece com as associações, você pode usar *sintaxe plugin* e fornecer opções de configuração adicionais:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'modified_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Core Behaviors

CounterCache

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹¹² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Timestamp

Nota: Atualmente, a documentação desta página não é suportada em português.

¹¹² <https://github.com/cakephp/docs>

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹¹³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Translate

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹¹⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Tree

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹¹⁵ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Criando Behavior

Nos exemplos a seguir, criaremos um bem simples `SluggableBehavior`. Esse behavior nos permitirá preencher um campo slug com o resultado de `Text::slug()` baseado em outro campo.

Antes de criar nosso behavior, devemos entender as convenções para behaviors:

- Behavior estão localizados em **src/Model/Behavior**, ou `MyPlugin\Model\Behavior`.
- Classes de Behavior devem estar no namespace `App\Model\Behavior`, ou no namespace `MyPlugin\Model\Behavior`.
- Classes de Behavior terminam com `Behavior`.
- Behaviors estendem `Cake\ORM\Behavior`.

Para criar nosso behavior sluggable. Coloque o seguinte em **src/Model/Behavior/SluggableBehavior.php**:

```
namespace App\Model\Behavior;

use Cake\ORM\Behavior;

class SluggableBehavior extends Behavior
```

¹¹³ <https://github.com/cakephp/docs>

¹¹⁴ <https://github.com/cakephp/docs>

¹¹⁵ <https://github.com/cakephp/docs>

```
{
}
```

Semelhante a classes de tabela (table classes), behaviors também tem um método `initialize()` onde você pode colocar o código de inicialização do seu behavior, se necessário:

```
public function initialize(array $config)
{
    // Algum código de inicialização aqui
}
```

Agora nós podemos adicionar esse behavior a uma de nossas classes de tabela (table classes). Neste exemplo, nós usaremos um `ArticlesTable`, pois artigos normalmente tem propriedades de slug para criar URLs amigáveis:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Sluggable');
    }
}
```

Nosso novo behavior não faz muita coisa no momento. Em seguida, iremos adicionar um método de mixin e um event listener para que, quando salvamos entidades nós podemos realizar slug automaticamente de um campo.

Definindo Métodos de Mixin

Qualquer método público definido em um behavior será adicionado como um método ‘mixin’ no objeto de tabela que está anexado. Se você anexar dois behavior que fornecem os mesmos métodos uma exceção será lançada. Se um behavior fornecer o mesmo método que uma classe de tabela, o método de behavior não será chamado pela tabela. Os métodos de mixin receberão exatamente os mesmos argumentos fornecidos à tabela. Por exemplo, se o nosso `SluggableBehavior` definiu o seguinte método:

```
public function slug($value)
{
    return Text::slug($value, $this->_config['replacement']);
}
```

Isto poderia ser invocado usando:

```
$slug = $articles->slug('My article name');
```

Limitando ou Renomeando Métodos de Mixin Expostos

Ao criar behaviors, pode haver situações em que você não deseja expor métodos públicos como métodos de ‘mixin’. Nesses casos, você pode usar a chave de configuração `implementedMethods` para renomear ou excluir métodos de ‘mixin’. Por exemplo, se quisermos prefixar nosso método `slug()`, nós poderíamos fazer o seguinte:

```
protected $_defaultConfig = [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
];
```

Aplicando essa configuração deixará `slug()` como não callable, no entanto, ele adicionará um método ‘mixin’ `superSlug` à tabela. Notavelmente, se nosso behavior implementasse outros métodos públicos eles **não** estariam disponíveis como métodos ‘mixin’ com a configuração acima.

Desde que os métodos expostos são decididos por configuração, você também pode renomear/remover métodos de ‘mixin’ ao adicionar um behavior à tabela. Por exemplo:

```
// In a table's initialize() method.
$this->addBehavior('Sluggable', [
    'implementedMethods' => [
        'superSlug' => 'slug',
    ]
]);
```

Defining Event Listeners

Agora que nosso behavior tem um método de ‘mixin’ para campos de slug, nós podemos implementar um listener de callback para automaticamente gerar slug de um campo quando entidades são salvas. Nós também iremos modificar nosso método de slug para aceitar uma entidade ao invéz de apenas um valor simples. Nosso behavior agora deve parecer com:

```
namespace App\Model\Behavior;

use ArrayObject;
use Cake\Datasource\EntityInterface;
use Cake\Event\Event;
use Cake\ORM\Behavior;
use Cake\ORM\Entity;
use Cake\ORM\Query;
use Cake\Utility\Text;

class SluggableBehavior extends Behavior
{
    protected $_defaultConfig = [
        'field' => 'title',
        'slug' => 'slug',
        'replacement' => '-',
    ];

    public function slug(Entity $entity)
    {
        $config = $this->config();
        $value = $entity->get($config['field']);
        $entity->set($config['slug'], Text::slug($value, $config['replacement']));
    }

    public function beforeSave(Event $event, EntityInterface $entity, ArrayObject
↵$options)
    {
        $this->slug($entity);
    }
}
```



```

    }
}

```

O código acima mostra alguns recursos interessantes de behaviors:

- Behaviors podem definir métodos de callback definindo métodos que seguem as convenções de *Lifecycle Callbacks*.
- Behaviors podem definir uma propriedade de configuração padrão. Essa propriedade é mesclada com as substituições quando um behavior é anexado à tabela.

Para evitar que o processo de gravação (save) continue, simplesmente pare a propagação do evento em seu callback:

```

public function beforeSave(Event $event, EntityInterface $entity, ArrayObject
    ↪ $options)
{
    if (...) {
        $event->stopPropagation();
        return;
    }
    $this->slug($entity);
}

```

Definindo Finders

Agora que somos capazes de salvar artigos com valores de slug, nós devemos implementar um método de ‘finder’(busca) para que possamos obter artigos por seus slugs. Em métodos de ‘finder’(busca) de behaviors, use as mesmas convenções que *Personalizando Métodos de Consulta* usa. Nosso método `find('slug')` pareceria com:

```

public function findSlug(Query $query, array $options)
{
    return $query->where(['slug' => $options['slug']]);
}

```

Uma vez que nosso behavior tem o método acima nós podemos chamá-lo:

```

$article = $articles->find('slug', ['slug' => $value])->first();

```

Limitando ou Renomeando Métodos Finder Expostos

Ao criar behaviors, pode haver situações em que você não deseja expor métodos finder, ou você precisa renomear o finder para evitar métodos duplicados. Nesses casos, você pode usar a chave de configuração `implementedFinders` para renomear ou excluir métodos finder. Por exemplo, se quisermos renomear nosso método `find(slug)`, nós poderíamos fazer o seguinte:

```

protected $_defaultConfig = [
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
];

```

Aplicando esta configuração fará com que `find('slug')` dispare um erro. No entanto, ela deixará disponível `find('slugged')`. Notavelmente, se nosso behavior implementasse outros métodos finder, eles **não** estariam disponíveis, pois não estão incluídos na configuração.

Desde que os métodos expostos são decididos por configuração, você também pode renomear/remover métodos finder ao adicionar um behavior à tabela. Por exemplo:

```
// No método initialize() da tabela.
$this->addBehavior('Sluggable', [
    'implementedFinders' => [
        'slugged' => 'findSlug',
    ]
]);
```

Transforming Request Data into Entity Properties

Behaviors podem definir lógica para como os campos personalizados que eles fornecem são arrumados (marshalled) implementando a `Cake\ORM\PropertyMarshalInterface`. Esta interface requer um único método para ser implementado:

```
public function buildMarshalMap($marshaller, $map, $options)
{
    return [
        'custom_behavior_field' => function ($value, $entity) {
            // Transform the value as necessary
            return $value . '123';
        }
    ];
}
```

O `TranslateBehavior` tem uma implementação não trivial desta interface à qual você pode querer referir.

Novo na versão 3.3.0: A capacidade de behaviors para participar do processo de marshalling foi adicionada em in 3.3.0

Removendo Behaviors Carregados

Para remover um behavior da sua tabela, você pode chamar o método `removeBehavior()`:

```
// Remove the loaded behavior
$this->removeBehavior('Sluggable');
```

Acessando Behaviors Carregados

Uma vez que você anexou behaviors à sua instância da `Table` você pode conferir os behaviors carregados ou acessar behaviors específicos usando o `BehaviorRegistry`:

```
// Verifica quais behaviors estão carregados
$table->behaviors()->loaded();

// Verifica se um behavior especifico está carregado
// Lembre-se de omitir o prefixo de plugin.
$table->behaviors()->has('CounterCache');

// Obtem um behavior carregado
// Lembre-se de omitir o prefixo de plugin
$table->behaviors()->get('CounterCache');
```

Re-configurando Behaviors Carregados

Para modificar a configuração de um behavior já carregado, você pode combinar o comando `BehaviorRegistry::get` com o comando `config` fornecido pela trait `InstanceConfigTrait`.

Por exemplo, se uma classe pai (por exemplo uma, `AppTable`) carregasse o behavior `Timestamp`, você poderia fazer o seguinte para adicionar, modificar ou remover as configurações do behavior. Nesse caso, nós adicionaremos um evento que queremos que o `Timestamp` responda:

```
namespace App\Model\Table;

use App\Model\Table\AppTable; // similar to AppController

class UsersTable extends AppTable
{
    public function initialize(array $options)
    {
        parent::initialize($options);

        // e.g. if our parent calls $this->addBehavior('Timestamp');
        // and we want to add an additional event
        if ($this->behaviors()->has('Timestamp')) {
            $this->behaviors()->get('Timestamp')->config([
                'events' => [
                    'Users.login' => [
                        'last_login' => 'always'
                    ],
                ],
            ]);
        }
    }
}
```

Schema

O CakePHP possui um sistema de schema que é capaz de refletir e gerar informações de schema para tabelas em SQL datastores. O sistema de schema pode gerar/refletir schema para qualquer plataforma que o CakePHP suporte.

As partes principais do sistema de schema são `Cake\Database\Schema\Collection` e `Cake\Database\Schema\TableSchema`. Essas classes te oferecem acesso a todo o banco de dados e recursos de tabela individual respectivamente.

O uso primário de sistema de schema é para *Fixtures*. No entanto, isso também ser usado em sua aplicação se requerido.

Objetos Schema\TableSchema

```
class Cake\Database\Schema\TableSchema
```

O subsistema de schema oferece um simples objeto `TableSchema` para guardar dados sobre uma tabela do banco de dados. Este objeto é retornado pelos recursos de reflexão de schema:

```
use Cake\Database\Schema\TableSchema;

// Criar uma tabela, uma coluna por vez.
$schema = new TableSchema('posts');
$schema->addColumn('id', [
```

```
'type' => 'integer',
'length' => 11,
'null' => false,
'default' => null,
])->addColumn('title', [
    'type' => 'string',
    'length' => 255,
    // Cria um campo de tamanho fixo (char field)
    'fixed' => true
])->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id']
]);

// Classes Schema\TableSchema também podem ser criados com array de dados
$schema = new TableSchema('posts', $columns);
```

Objetos `Schema\TableSchema` permitem você construir apartir de informações sobre schema de tabelas. Isto ajuda a normalizar e validar os dados usados para descrever uma tabela. Por exemplo, as duas formas a seguir são equivalentes:

```
$schema->addColumn('title', 'string');
// e
$schema->addColumn('title', [
    'type' => 'string'
]);
```

Enquanto equivalente, a 2ª forma permite mais detalhes e controle. Isso emula os recursos existentes disponíveis em arquivos de `Schema` + os `schema` de `fixture` em 2.x.

Acessando Dados de Coluna

Colunas são adicionadas como argumentos do construtor, ou pelo método `addColumn()`. Uma vez que os campos são adicionados, as informações podem ser obtidas usando o método `column()` ou `columns()`:

```
// Obtem um array de dados sobre a coluna
$c = $schema->column('title');

// Obtem uma lista com todas as colunas.
$cols = $schema->columns();
```

Índices e Restrições

Os índices são adicionado usando `addIndex()`. Restrições são adicionadas usando `addConstraint()`. Os índices e restrições não podem ser adicionados para colunas que não existem, já que isso resultaria em um estado inválido. Os índices são diferentes de restrições, e exceções serão disparadas se você tentar misturar tipos entre os métodos. Um exemplo de ambos os métodos é:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
->addColumn('author_id', 'integer')
->addColumn('title', 'string')
->addColumn('slug', 'string');
```

```
// Adiciona uma chave primária.
$schema->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id']
]);
// Adiciona uma chave única
$schema->addConstraint('slug_idx', [
    'columns' => ['slug'],
    'type' => 'unique',
]);
// Adiciona um index
$schema->addIndex('slug_title', [
    'columns' => ['slug', 'title'],
    'type' => 'index'
]);
// Adiciona uma chave estrangeira
$schema->addConstraint('author_id_idx', [
    'columns' => ['author_id'],
    'type' => 'foreign',
    'references' => ['authors', 'id'],
    'update' => 'cascade',
    'delete' => 'cascade'
]);
```

Se você adicionar uma restrição de chave primária para uma coluna do tipo integer, ela será automaticamente convertida em uma coluna auto-increment/serial dependendo da plataforma de banco de dados:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', 'integer')
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id']
]);
```

No exemplo abaixo a coluna `id` geraria o seguinte SQL em MySQL:

```
CREATE TABLE `posts` (
  `id` INTEGER AUTO_INCREMENT,
  PRIMARY KEY (`id`)
)
```

Se sua chave primária contém mais que uma coluna, nenhuma delas serão automaticamente convertidas para um valor auto-incremento. Em vez disso, você precisará dizer ao objeto da tabela qual coluna na chave composta que você deseja usar auto-incremento:

```
$schema = new TableSchema('posts');
$schema->addColumn('id', [
    'type' => 'integer',
    'autoIncrement' => true,
]);
->addColumn('account_id', 'integer')
->addConstraint('primary', [
    'type' => 'primary',
    'columns' => ['id', 'account_id']
]);
```

A opção `autoIncrement` apenas funciona com colunas do tipo `integer` e `biginteger`.

Lendo Índices e Restrições

Os índices e restrições podem ser lido de um objeto de tabela usando métodos acessores. Assumindo que `$schema` é uma instância de `TableSchema` populada, você poderia fazer o seguinte:

```
// Obter restrições. Retornará os
// nomes de todas as restrições.
$constraints = $schema->constraints();

// Obter dados sobre uma restrição.
$constraint = $schema->constraint('author_id_idx')

// Obter índices. Retornará os
// nomes de todos os índices
$indexes = $schema->indexes();

// Obter dados sobre um índice
$index = $schema->index('author_id_idx')
```

Adicionando Opções de Tabela

Alguns drivers (principalmente MySQL) suportam e requerem metadados de tabela adicionais. No caso do MySQL as propriedades `CHARSET`, `COLLATE` e `ENGINE` são requeridos para manter a estrutura de uma tabela no MySQL. O seguinte pode ser usado para adicionar opções de tabela:

```
$schema->options([
    'engine' => 'InnoDB',
    'collate' => 'utf8_unicode_ci',
]);
```

Os dialetos de plataforma apenas cuidam das chaves que eles estão interessados e ignoram o resto. Nem todas as opções são suportadas por todas as plataformas.

Convertendo TableSchema em SQL

Usando os métodos `createSql()` ou `dropSql()` você pode obter SQL específico de plataforma para criar ou remover uma tabela específica:

```
$db = ConnectionManager::get('default');
$schema = new TableSchema('posts', $fields, $indexes);

// Criar uma tabela
$queries = $schema->createSql($db);
foreach ($queries as $sql) {
    $db->execute($sql);
}

// Remover um tabela
$sql = $schema->dropSql($db);
$db->execute($sql);
```

Ao usar o driver de conexão, os dados de schema podem ser convertidos em SQL específico da plataforma. O retorno de `createSql` e `dropSql` é uma lista de consultas SQL requeridas para criar uma tabela e os índices. Algumas

plataformas podem requerer várias declarações para criar tabelas com comentários e/ou índices. Um array de consultas SQL é sempre retornado.

Schema Collections

class Cake\Database\Schema\Collection

Collection fornece acesso as várias tabelas disponíveis numa conexão. Você pode usar isto para obter a lista de tabelas ou refletir tabelas em objetos *TableSchema*. O uso básico da classe parece com:

```
$db = ConnectionManager::get('default');

// Criar uma coleção de schema.
$collection = $db->schemaCollection();

// Obtem os nomes das tabelas.
$tables = $collection->listTables();

// Obtem uma tabela específica (instância de Schema\TableSchema)
$tableSchema = $collection->describe('posts');
```

ORM Cache Shell

O OrmCacheShell fornece uma ferramenta CLI simples para gerenciar caches de metadados da sua aplicação. Em situações de implantação, é útil reconstruir o cache de metadados no local sem limpar os dados de cache existentes. Você pode fazer isso executando:

```
bin/cake orm_cache build --connection default
```

Isso irá reconstruir o cache de metadados para todas as tabelas na conexão default. Se você só precisa reconstruir uma única tabela, você pode fazer isso fornecendo seu nome:

```
bin/cake orm_cache build --connection default <<Nome>>
```

Além de criar dados em cache, você pode usar o OrmCacheShell para remover metadados em cache também:

```
# Limpar todos os metadados
bin/cake orm_cache clear

# Limpar uma única tabela de metadados
bin/cake orm_cache clear <<Nome>>
```

Bake Console

O bake console do CakePHP é outro empenho para você ter o CakePHP configurado e funcionando rápido. O bake console pode criar qualquer ingrediente básico do CakePHP: models, behaviors, views, helpers, components, test cases, fixtures e plugins. E nós não estamos apenas falando de classes esqueleto: O Bake pode criar uma aplicação totalmente funcional em questão de minutos. De fato, o Bake é um passo natural a se dar uma vez que a aplicação tem seu alicerce construído.

Instalação

Antes de tentar usar ou estender o bake, tenha certeza que ele está instalado em sua aplicação. O bake é distribuído como um plugin que você pode instalar com o Composer:

```
composer require --dev cakephp/bake:~1.0
```

Isto irá instalar o bake como uma dependência de desenvolvimento, sendo assim, não instalado quando em um ambiente de produção. As seções a seguir cobrem o uso do bake com mais detalhes:

Geração de código com bake

O console do cake é executado usando a CLI do PHP (interface da linha de comando). Se você tiver problemas para executar o script, assegure-se de que:

1. Você instalou a CLI do PHP e possui os módulos apropriados habilitados (por exemplo: MySQL, intl).
2. Os usuários também podem ter problemas se o host do banco de dados for 'localhost' e deve tentar '127.0.0.1', em vez disso, como localhost pode causar problemas com CLI PHP.
3. Dependendo de como o seu computador está configurado, você pode ter que definir direitos de execução no script cake bash para chamá-lo usando `bin/cake bake`.

Antes de executar o Bake você deve certificar-se de ter pelo menos um banco de dados conexão configurada. Veja a seção sobre [database configuration](#) para mais informações.

Quando executado sem argumentos `bin/cake bake` irá exibir uma lista de tarefas. Você deve ver algo como:

```
$ bin/cake bake
```

```
Welcome to CakePHP v3.4.6 Console
-----
App : src
Path: /var/www/cakephp.dev/src/
PHP : 5.6.20
-----

The following commands can be used to generate skeleton code for your application.

Available bake commands:

- all
- behavior
- cell
- component
- controller
- fixture
- form
- helper
- mailer
- migration
- migration_diff
- migration_snapshot
- model
- plugin
- seed
- shell
- shell_helper
- task
- template
- test

By using `cake bake [name]` you can invoke a specific bake task.
```

Você pode obter mais informações sobre o que cada tarefa faz e quais são as opções disponível usando o `--help` option:

```
$ bin/cake bake --help

Welcome to CakePHP v3.4.6 Console
-----
App : src
Path: /var/www/cakephp.dev/src/
PHP : 5.6.20
-----

The Bake script generates controllers, models and template files for
your application. If run with no command line arguments, Bake guides the
user through the class creation process. You can customize the
generation process by telling Bake where different parts of your
application are using command line arguments.

Usage:
cake bake.bake [subcommand] [options]

Subcommands:

all                Bake a complete MVC skeleton.
behavior           Bake a behavior class file.
cell              Bake a cell class file.
```

```

component      Bake a component class file.
controller     Bake a controller skeleton.
fixture        Generate fixtures for use with the test suite. You
               can use `bake fixture all` to bake all fixtures.

form           Bake a form class file.
helper         Bake a helper class file.
mailer         Bake a mailer class file.
migration      Bake migration class.
migration_diff Bake migration class.
migration_snapshot Bake migration snapshot class.
model          Bake table and entity classes.
plugin         Create the directory structure, AppController class
               and testing setup for a new plugin. Can create
               plugins in any of your bootstrapped plugin paths.

seed           Bake seed class.
shell          Bake a shell class file.
shell_helper   Bake a shell_helper class file.
task           Bake a task class file.
template       Bake views for a controller, using built-in or
               custom templates.
test           Bake test case skeletons for classes.

```

To see help on a subcommand **use** ``cake bake.bake [subcommand] --help``

Options:

```

--connection, -c Database connection to use in conjunction with `bake
all`. (default: default)
--everything      Bake a complete MVC skeleton, using all the available
                 tables. Usage: "bake all --everything"
--force, -f       Force overwriting existing files without prompting.
--help, -h        Display this help.
--plugin, -p      Plugin to bake into.
--prefix          Prefix to bake controllers and templates into.
--quiet, -q       Enable quiet output.
--tablePrefix     Table prefix to be used in models.
--theme, -t       The theme to use when baking code. (choices:
                 Bake|Migrations)
--verbose, -v     Enable verbose output.

```

Temas para o Bake

A opção de tema é comum a todos os comandos do bake, e permite mudar os arquivos de modelo usados no bake. Para criar seus próprios modelos, veja a *documentação de criação de themes para o bake*.

Extendendo o Bake

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹¹⁶ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

¹¹⁶ <https://github.com/cakephp/docs>

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Caching

Configuring Cache Class

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹¹⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹¹⁷ <https://github.com/cakephp/docs>

Console e Shells

O CakePHP não oferece um framework apenas para desenvolvimento web, mas também um framework para criação de aplicações de console. Estas aplicações são ideais para manipular variadas tarefas em segundo plano como manutenção e complementação de trabalho fora do ciclo requisição-resposta. As aplicações de console do CakePHP permitem a você reutilizar suas classes de aplicação a partir da linha de comando.

O CakePHP traz consigo algumas aplicações de console nativas. Algumas dessas aplicações são utilizadas em conjunto com outros recursos do CakePHP (como `i18n`), e outros de uso geral para aceleração de trabalho.

O Console do CakePHP

Esta seção provê uma introdução à linha de comando do CakePHP. Ferramentas de console são ideais para uso em cron jobs, ou utilitários baseados em linha de comando que não precisam ser acessíveis por um navegador web.

O PHP provê um cliente CLI que faz interface com o seu sistema de arquivos e aplicações de forma muito mais suave. O console do CakePHP provê um framework para criar scripts shell. O console utiliza uma configuração tipo dispatcher para carregar uma shell ou tarefa, e prover seus parâmetros.

Nota: Uma linha de comando (CLI) constituída a partir do PHP deve estar disponível no sistema se você planeja utilizar o Console.

Antes de entrar em detalhes, vamos ter certeza de que você pode executar o console do CakePHP. Primeiro, você vai precisar executar um sistema shell. Os exemplos apresentados nesta seção serão em bash, mas o Console do CakePHP é compatível com o Windows também. Este exemplo assume que o usuário está conectado em um prompt do bash e está atualmente na raiz de uma aplicação CakePHP.

Aplicações CakePHP possuem um diretório *Console* que contém todas as shells e tarefas para uma aplicação. Ele também vem com um executável:

```
$ cd /path/to/app
$ bin/cake
```

Executar o Console sem argumentos produz esta mensagem de ajuda:

```
Welcome to CakePHP v3.0.0 Console
-----
```

```
App : App
Path: /Users/markstory/Sites/cakephp-app/src/
-----
Current Paths:

-app: src
-root: /Users/markstory/Sites/cakephp-app
-core: /Users/markstory/Sites/cakephp-app/vendor/cakephp/cakephp

Changing Paths:

Your working path should be the same as your application path. To change your path,
↪ use the '-app' param.
Example: -app relative/path/to/myapp or -app /absolute/path/to/myapp

Available Shells:

[Bake] bake

[Migrations] migrations

[CORE] i18n, orm_cache, plugin, server

[app] behavior_time, console, orm

To run an app or core command, type cake shell_name [args]
To run a plugin command, type cake Plugin.shell_name [args]
To get help on a specific command, type cake shell_name --help
```

A primeira informação impressa refere-se a caminhos. Isso é útil se você estiver executando o console a partir de diferentes partes do sistema de arquivos.

Criando uma Shell

Vamos criar uma shell para utilizar no Console. Para este exemplo, criaremos uma simples Hello World (Olá Mundo) shell. No diretório **src/Shell** de sua aplicação crie **HelloShell.php**. Coloque o seguinte código dentro do arquivo recém criado:

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
```

As convenções para as classes de shell são de que o nome da classe deve corresponder ao nome do arquivo, com o sufixo de Shell. No nosso shell criamos um método `main()`. Este método é chamado quando um shell é chamado sem comandos adicionais. Vamos adicionar alguns comandos daqui a pouco, mas por agora vamos executar a nossa shell. A partir do diretório da aplicação, execute:


```
bin/cake hello
```

Você deve ver a seguinte saída:

```
Welcome to CakePHP Console
-----
App : app
Path: /Users/markstory/Sites/cake_dev/src/
-----
Hello world.
```

Como mencionado antes, o método `main()` em shells é um método especial chamado sempre que não há outros comandos ou argumentos dados para uma shell. Por nosso método principal não ser muito interessante, vamos adicionar outro comando que faz algo:

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
    public function main()
    {
        $this->out('Hello world.');
```

Depois de salvar o arquivo, você deve ser capaz de executar o seguinte comando e ver o seu nome impresso:

```
bin/cake hello hey_there your-name
```

Qualquer método público não prefixado por um `_` é permitido para ser chamado a partir da linha de comando. Como você pode ver, os métodos invocados a partir da linha de comando são transformados do argumento prefixado para a forma correta do nome camel-cased (camelizada) na classe.

No nosso método `heyThere()` podemos ver que os argumentos posicionais são providos para nossa função `heyThere()`. Argumentos posicionais também estão disponíveis na propriedade `args`. Você pode acessar switches ou opções em aplicações shell, estando disponíveis em `$this->params`, mas nós iremos cobrir isso daqui a pouco.

Quando utilizando um método `main()` você não estará liberado para utilizar argumentos posicionais. Isso se deve ao primeiro argumento posicional ou opção ser interpretado(a) como o nome do comando. Se você quer utilizar argumentos, você deve usar métodos diferentes de `main()`.

Usando Models em suas shells

Você frequentemente precisará acessar a camada lógica de negócios em seus utilitários shell; O CakePHP faz essa tarefa super fácil. Você pode carregar models em shells assim como faz em um controller utilizando `loadModel()`. Os models carregados são definidos como propriedades anexas à sua shell:

```
namespace App\Shell;
```

```
use Cake\Console\Shell;

class UserShell extends Shell
{
    public function initialize()
    {
        parent::initialize();
        $this->loadModel('Users');
    }

    public function show()
    {
        if (empty($this->args[0])) {
            return $this->error('Por favor, indique um nome de usuário.');
```

A shell acima, irá preencher um user pelo seu username e exibir a informação armazenada no banco de dados.

Tasks de Shell

Haverão momentos construindo aplicações mais avançadas de console que você vai querer compor funcionalidades em classes reutilizáveis que podem ser compartilhadas através de muitas shells. Tasks permitem que você extraia comandos em classes. Por exemplo, o `bake` é feito quase que completamente de tasks. Você define tasks para uma shell usando a propriedade `$tasks`:

```
class UserShell extends Shell
{
    public $tasks = ['Template'];
}
```

Você pode utilizar tasks de plugins utilizando o padrão *sintaxe plugin*. Tasks são armazenadas sob `Shell/Task/` em arquivos nomeados depois de suas classes. Então se nós estivéssemos criando uma nova task 'FileGenerator', você deveria criar `src/Shell/Task/FileGeneratorTask.php`.

Cada task deve ao menos implementar um método `main()`. O `ShellDispatcher`, vai chamar esse método quando a task é invocada. Uma classe task se parece com:

```
namespace App\Shell\Task;

use Cake\Console\Shell;

class FileGeneratorTask extends Shell
{
    public function main()
    {
    }
}
```

Uma shell também pode prover acesso a suas tasks como propriedades, que fazem tasks serem ótimas para criar punhados de funcionalidade reutilizáveis similares a *Components (Componentes)*:

```
// Localizado em src/Shell/SeaShell.php
class SeaShell extends Shell
{
    // Localizado em src/Shell/Task/SoundTask.php
    public $tasks = ['Sound'];

    public function main()
    {
        $this->Sound->main();
    }
}
```

Você também pode acessar tasks diretamente da linha de comando:

```
$ cake sea sound
```

Nota: Para acessar tasks diretamente através da linha de comando, a task **deve** ser incluída na propriedade da classe shell \$tasks. Portanto, esteja ciente que um método chamado “sound” na classe SeaShell deve sobrescrever a habilidade de acessar a funcionalidade na task Sound, especificada no array \$tasks.

Carregando Tasks em tempo-real com TaskRegistry

Você pode carregar arquivos em tempo-real utilizando o Task registry object. Você pode carregar tasks que não foram declaradas no \$tasks dessa forma:

```
$project = $this->Tasks->load('Project');
```

Carregará e retornará uma instância ProjectTask. Você pode carregar tasks de plugins usando:

```
$progressBar = $this->Tasks->load('ProgressBar.ProgressBar');
```

Invocando outras Shells a partir da sua Shell

Cake\Console\dispatchShell(\$args)

Existem ainda muitos casos onde você vai querer invocar uma shell a partir de outra. Shell::dispatchShell() lhe dá a habilidade de chamar outras shells ao providenciar o argv para a sub shell. Você pode providenciar argumentos e opções tanto como variáveis ou como strings:

```
// Como uma string
$this->dispatchShell('schema create Blog --plugin Blog');

// Como um array
$this->dispatchShell('schema', 'create', 'Blog', '--plugin', 'Blog');
```

O conteúdo acima mostra como você pode chamar a shell schema para criar o schema de um plugin de dentro da shell do próprio.

Recenendo Input de usuários

`Cake\Console\in($question, $choices = null, $default = null)`

Quando construir aplicações interativas pelo console você irá precisar receber inputs dos usuários. CakePHP oferece uma forma fácil de fazer isso:

```
// Receber qualquer texto dos usuários.
$color = $this->in('What color do you like?');

// Receber uma escolha dos usuários.
$selection = $this->in('Red or Green?', ['R', 'G'], 'R');
```

A validação de seleção é insensitiva a maiúsculas / minúsculas.

Criando Arquivos

`Cake\Console\createFile($path, $contents)`

Muitas aplicações Shell auxiliam tarefas de desenvolvimento e implementação. Criar arquivos é frequentemente importante nestes casos de uso. O CakePHP oferece uma forma fácil de criar um arquivo em um determinado diretório:

```
$this->createFile('bower.json', $stuff);
```

Se a Shell for interativa, um alerta vai ser gerado, e o usuário questionado se ele quer sobrescrever o arquivo caso já exista. Se a propriedade de interação da shell for `false`, nenhuma questão será disparada e o arquivo será simplesmente sobrescrito.

Saída de dados do Console

A classe `Shell` oferece alguns métodos para direcionar conteúdo:

```
// Escreve para stdout
$this->out('Normal message');

// Escreve para stderr
$this->err('Error message');

// Escreve para stderr e para o processo
$this->error('Fatal error');
```

A Shell também inclui métodos para limpar a saída de dados, criando linhas em branco, ou desenhando uma linha de traços:

```
// Exibe 2 linhas novas
$this->out($this->nl(2));

// Limpa a tela do usuário
$this->clear();

// Desenha uma linha horizontal
$this->hr();
```

Por último, você pode atualizar a linha atual de texto na tela usando `_io->overwrite()`:

```
$this->out('Counting down');
$this->out('10', 0);
for ($i = 9; $i > 0; $i--) {
    sleep(1);
    $this->_io->overwrite($i, 0, 2);
}
```

É importante lembrar, que você não pode sobrescrever texto uma vez que uma nova linha tenha sido exibida.

Console Output Levels

Shells frequentemente precisam de diferentes níveis de verbosidade. Quando executadas como cron jobs, muitas saídas são desnecessárias. E há ocasiões que você não estará interessado em tudo que uma shell tenha a dizer. Você pode usar os níveis de saída para sinalizar saídas apropriadamente. O usuário da shell, pode então decidir qual nível de detalhe ele está interessado ao sinalizar o chamado da shell. `Cake\Console\Shell::out()` suporta 3 tipos de saída por padrão.

- QUIET - Apenas informação absolutamente importante deve ser sinalizada.
- NORMAL - O nível padrão, e uso normal.
- VERBOSE - Sinalize mensagens que podem ser irritantes em demasia para uso diário, mas informativas para depuração como VERBOSE.

Você pode sinalizar a saída da seguinte forma:

```
// Deve aparecer em todos os níveis.
$this->out('Quiet message', 1, Shell::QUIET);
$this->quiet('Quiet message');

// Não deve aparecer quando a saída quiet estiver alternado.
$this->out('normal message', 1, Shell::NORMAL);
$this->out('loud message', 1, Shell::VERBOSE);
$this->verbose('Verbose output');

// Deve aparecer somente quando a saída verbose estiver habilitada.
$this->out('extra message', 1, Shell::VERBOSE);
$this->verbose('Verbose output');
```

Você pode controlar o nível de saída das shells, ao usar as opções `--quiet` e `--verbose`. Estas opções são adicionadas por padrão, e permitem a você controlar consistentemente níveis de saída dentro das suas shells do CakePHP.

Estilizando a saída de dados

Estilizar a saída de dados é feito ao incluir tags - como no HTML - em sua saída. O `ConsoleOutput` irá substituir estas tags com a sequência correta de código ansi. Hão diversos estilos nativos, e você pode criar mais. Os nativos são:

- `error` Mensagens de erro. Texto sublinhado vermelho.
- `warning` Mensagens de alerta. Texto amarelo.
- `info` Mensagens informativas. Texto ciano.
- `comment` Texto adicional. Texto azul.
- `question` Texto que é uma questão, adicionado automaticamente pela shell.

Você pode criar estilos adicionais usando `$this->stdout->styles()`. Para declarar um novo estilo de saída você pode fazer:

```
$this->_io->styles('flashy', ['text' => 'magenta', 'blink' => true]);
```

Isso deve então permiti-lo usar uma `<flashy>` tag na saída de sua shell, e se as cores ansi estiverem habilitadas, o seguinte pode ser renderizado como texto magenta piscante `$this->out('<flashy>Whooaa</flashy> Something went wrong');`. Quando definir estilos você pode usar as seguintes cores para os atributos `text` e `background`:

- black
- red
- green
- yellow
- blue
- magenta
- cyan
- white

Você também pode usar as seguintes opções através de valores booleanos, defini-los com valor positivo os habilita.

- bold
- underline
- blink
- reverse

Adicionar um estilo o torna disponível para todas as instâncias do `ConsoleOutput`, então você não tem que redeclarar estilos para os objetos `stdout` e `stderr` respectivamente.

Desabilitando a colorização

Mesmo que a colorização seja incrível, haverá ocasiões que você querará desabilitá-la, ou forçá-la:

```
$this->_io->outputAs(ConsoleOutput::RAW);
```

O citado irá colocar o objeto de saída em modo raw. Em modo raw, nenhum estilo é aplicado. Existem três modos que você pode usar.

- `ConsoleOutput::RAW` - Saída raw, nenhum estilo ou formatação serão aplicados. Este é um modo indicado se você estiver exibindo XML ou, quiser depurar porquê seu estilo não está funcionando.
- `ConsoleOutput::PLAIN` - Saída de texto simples, tags conhecidas de estilo serão removidas da saída.
- `ConsoleOutput::COLOR` - Saída onde a cor é removida.

Por padrão em sistemas `*nix` objetos `ConsoleOutput` padronizam-se a a saída de cores. Em sistemas Windows, a saída simples é padrão a não ser que a variável de ambiente `ANSICON` esteja presente.

Opções de configuração e Geração de ajuda

```
class Cake\Console\ConsoleOptionParser
```

ConsoleOptionParser oferece uma opção de CLI e analisador de argumentos.

OptionParsers permitem a você completar dois objetivos ao mesmo tempo. Primeiro, eles permitem definir opções e argumentos para os seus comandos. Isso permite separar validação básica de dados e seus comandos do console. Segundo, permite prover documentação, que é usada para gerar arquivos de ajuda bem formatados.

O console framework no CakePHP recebe as opções do seu interpretador shell ao chamar `$this->getOptionParser()`. Sobre-escrever esse método permite configurar o OptionParser para definir as entradas aguardadas da sua shell. Você também pode configurar interpretadores de subcomandos, que permitem ter diferentes interpretadores para subcomandos e tarefas. O ConsoleOptionParser implementa uma interface fluida e inclui métodos para facilmente definir múltiplas opções/argumentos de uma vez:

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    // Configure parser
    return $parser;
}
```

Configurando um interpretador de opção com a interface fluida

Todos os métodos que configuram um interpretador de opções podem ser encadeados, permitindo definir um interpretador de opções completo em uma série de chamadas de métodos:

```
public function getOptionParser()
{
    $parser = parent::getOptionParser();
    $parser->addArgument('type', [
        'help' => 'Either a full path or type of class.'
    ])->addArgument('className', [
        'help' => 'A CakePHP core class name (e.g: Component, HtmlHelper).'
    ])->addOption('method', [
        'short' => 'm',
        'help' => __('The specific method you want help on.')
    ])->description(__('Lookup doc block comments for classes in CakePHP.'));
    return $parser;
}
```

Os métodos que permitem encadeamento são:

- `description()`
- `epilog()`
- `command()`
- `addArgument()`
- `addArguments()`
- `addOption()`
- `addOptions()`
- `addSubcommand()`
- `addSubcommands()`

`Cake\Console\ConsoleOptionParser::description($text = null)`

Recebe ou define a descrição para o interpretador de opções. A descrição é exibida acima da informação do argumento e da opção. Ao instanciar tanto em array como em string, você pode definir o valor da descrição. Instanciar sem argumentos vai retornar o valor atual:

```
// Define múltiplas linhas de uma vez
$parser->description(['line one', 'line two']);

// Lê o valor atual
$parser->description();
```

`Cake\Console\ConsoleOptionParser::epilog($text = null)`

Recebe ou define o epílogo para o interpretador de opções. O epílogo é exibido depois da informação do argumento e da opção. Ao instanciar tanto em array como em string, você pode definir o valor do epílogo. Instanciar sem argumentos vai retornar o valor atual:

```
// Define múltiplas linhas de uma vez
$parser->epilog(['line one', 'line two']);

// Lê o valor atual
$parser->epilog();
```

Adicionando argumentos

`Cake\Console\ConsoleOptionParser::addArgument($name, $params = [])`

Argumentos posicionais são frequentemente usados em ferramentas de linha de comando, e `ConsoleOptionParser` permite definir argumentos bem como torná-los requiríveis. Você pode adicionar argumentos um por vez com `$parser->addArgument()`; ou múltiplos de uma vez com `$parser->addArguments()`;

```
$parser->addArgument('model', ['help' => 'The model to bake']);
```

Você pode usar as seguintes opções ao criar um argumento:

- **help** O texto de ajuda a ser exibido para este argumento.
- **required** Se esse parâmetro é requisito.
- **index** O índice do argumento, se deixado indefinido, o argumento será colocado no final dos argumentos. Se você definir o mesmo índice duas vezes, a primeira opção será sobreescrita.
- **choices** Um array de opções válidas para esse argumento. Se deixado vazio, todos os valores são válidos. Uma exceção será lançada quando `parse()` encontrar um valor inválido.

Argumentos que forem definidos como requisito lançarão uma exceção quando interpretarem o comando se eles forem omitidos. Então você não tem que lidar com isso em sua shell.

`Cake\Console\ConsoleOptionParser::addArguments(array $args)`

Se você tem um array com múltiplos argumentos você pode usar `$parser->addArguments()` para adicioná-los de uma vez.:

```
$parser->addArguments([
    'node' => ['help' => 'The node to create', 'required' => true],
    'parent' => ['help' => 'The parent node', 'required' => true]
]);
```


Assim como todos os métodos de construção no `ConsoleOptionParser`, `addArguments` pode ser usado como parte de um fluido método encadeado.

Validando argumentos

Ao criar argumentos posicionais, você pode usar a marcação `required` para indicar que um argumento deve estar presente quando uma shell é chamada. Adicionalmente você pode usar o `choices` para forçar um argumento a ser de uma lista de escolhas válidas:

```
$parser->addArgument('type', [
    'help' => 'The type of node to interact with.',
    'required' => true,
    'choices' => ['aro', 'aco']
]);
```

O código acima irá criar um argumento que é requisitado e tem validação no input. Se o argumento está tanto indefinido, ou possui um valor incorreto, uma exceção será lançada e a shell parará.

Adicionando opções

`Cake\Console\ConsoleOptionParser::addOption($name, $options = [])`

Opções são frequentemente usadas em ferramentas CLI. `ConsoleOptionParser` suporta a criação de opções com `verbose` e aliases curtas, suprimindo padrões e criando ativadores booleanos. Opções são criadas tanto com `$parser->addOption()` ou `$parser->addOptions()`:

```
$parser->addOption('connection', [
    'short' => 'c',
    'help' => 'connection',
    'default' => 'default',
]);
```

O código citado permite a você usar tanto `cake myshell --connection=other`, `cake myshell --connection other`, ou `cake myshell -c other` quando invocando a shell. Você também criar ativadores booleanos. Estes ativadores não consomem valores, e suas presenças apenas os habilitam nos parâmetros interpretados.:

```
$parser->addOption('no-commit', ['boolean' => true]);
```

Com essa opção, ao chamar uma shell como `cake myshell --no-commit something` o parâmetro `no-commit` deve ter um valor de `true`, e *something* deve ser tratado como um argumento posicional. As opções nativas `--help`, `--verbose`, e `--quiet` usam essa funcionalidade.

Ao criar opções você pode usar os seguintes argumentos para definir o seu comportamento:

- `short` - A variação de letra única para essa opção, deixe indefinido para `none`.
- `help` - Texto de ajuda para essa opção. Usado ao gerar ajuda para a opção.
- `default` - O valor padrão para essa opção. Se não estiver definido o valor padrão será `true`.
- `boolean` - A opção não usa valor, é apenas um ativador booleano. Por padrão `false`.
- `choices` - Um array de escolhas válidas para essa opção. Se deixado vazio, todos os valores são considerados válidos. Uma exceção será lançada quando `parse()` encontrar um valor inválido.

`Cake\Console\ConsoleOptionParser::addOptions(array $options)`

Se você tem um array com múltiplas opções, você pode usar `$parser->addOptions()` para adicioná-las de uma vez.:

```
$parser->addOptions([
    'node' => ['short' => 'n', 'help' => 'The node to create'],
    'parent' => ['short' => 'p', 'help' => 'The parent node']
]);
```

Assim como com todos os métodos construtores, no `ConsoleOptionParser`, `addOptions` pode ser usado como parte de um método fluente encadeado.

Validando opções

Opções podem ser fornecidas com um conjunto de escolhas bem como argumentos posicionais podem ser. Quando uma opção define escolhas, essas são as únicas opções válidas para uma opção. Todos os outros valores irão gerar um `InvalidArgumentException`:

```
$parser->addOption('accept', [
    'help' => 'What version to accept.',
    'choices' => ['working', 'theirs', 'mine']
]);
```

Usando opções booleanas

As opções podem ser definidas como opções booleanas, que são úteis quando você precisa criar algumas opções de marcação. Como opções com padrões, opções booleanas sempre irão incluir `-se` nos parâmetros analisados. Quando as marcações estão presentes elas são definidas para `true`, quando elas estão ausentes, são definidas como `false`:

```
$parser->addOption('verbose', [
    'help' => 'Enable verbose output.',
    'boolean' => true
]);
```

A opção seguinte resultaria em `$this->params['verbose']` sempre estando disponível. Isso permite a você omitir verificações `empty()` ou `isset()` em marcações booleanas:

```
if ($this->params['verbose']) {
    // Do something.
}
```

Desde que as opções booleanas estejam sempre definidas como `true` ou `false`, você pode omitir métodos de verificação adicionais.

Adicionando subcomandos

`Cake\Console\ConsoleOptionParser::addSubcommand($name, $options = [])`

Aplicativos de console são muitas vezes feitas de subcomandos, e esses subcomandos podem exigir a análise de opções especiais e terem a sua própria ajuda. Um perfeito exemplo disso é `bake`. `Bake` é feita de muitas tarefas separadas e todas têm a sua própria ajuda e opções. `ConsoleOptionParser` permite definir subcomandos e fornecer comandos analisadores de opção específica, de modo que a shell sabe como analisar os comandos para as suas funções:

```
$parser->addSubcommand('model', [
    'help' => 'Bake a model',
    'parser' => $this->Model->getOptionParser()
]);
```

A descrição acima é um exemplo de como você poderia fornecer ajuda e um especializado interpretador de opção para a tarefa de uma shell. Ao chamar a tarefa de `getOptionParser()` não temos de duplicar a geração do interpretador de opção, ou misturar preocupações no nosso shell. Adicionar subcomandos desta forma tem duas vantagens. Primeiro, ele permite que o seu shell documente facilmente seus subcomandos na ajuda gerada. Ele também dá fácil acesso ao subcomando `help`. Com o subcomando acima criado você poderia chamar `cake myshell --help` e ver a lista de subcomandos, e também executar o `cake myshell model --help` para exibir a ajuda apenas o modelo de tarefa.

Nota: Uma vez que seu Shell define subcomandos, todos os subcomandos deve ser explicitamente definidos.

Ao definir um subcomando, você pode usar as seguintes opções:

- `help` - Texto de ajuda para o subcomando.
- `parser` - Um `ConsoleOptionParser` para o subcomando. Isso permite que você crie métodos analisadores de opção específicos. Quando a ajuda é gerada por um subcomando, se um analisador está presente ele vai ser usado. Você também pode fornecer o analisador como uma matriz que seja compatível com `Cake\Console\ConsoleOptionParser::buildFromArray()`

Adicionar subcomandos pode ser feito como parte de uma cadeia de métodos fluente.

Construir uma ConsoleOptionParser de uma matriz

`Cake\Console\ConsoleOptionParser::buildFromArray($spec)`

Como mencionado anteriormente, ao criar interpretadores de opção de subcomando, você pode definir a especificação interpretadora como uma matriz para esse método. Isso pode ajudar fazer analisadores mais facilmente, já que tudo é um array:

```
$parser->addSubcommand('check', [
    'help' => __('Check the permissions between an ACO and ARO.'),
    'parser' => [
        'description' => [
            __('Use this command to grant ACL permissions. Once executed, the '),
            __('ARO specified (and its children, if any) will have ALLOW access '),
            __('to the specified ACO action (and the ACO's children, if any).')
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]
]);
```

Dentro da especificação do interpretador, você pode definir as chaves para `arguments`, `options`, `description` e `epilog`. Você não pode definir subcommands dentro de um construtor estilo array. Os valores para os argumentos e opções, devem seguir o formato que `Cake\Console\ConsoleOptionParser::addArguments()` e `Cake\Console\ConsoleOptionParser::addOptions()` usam. Você também pode usar `buildFromArray` por conta própria, para construir um interpretador de opção:

```
public function getOptionParser()
{
    return ConsoleOptionParser::buildFromArray([
        'description' => [
            __("Use this command to grant ACL permissions. Once executed, the "),
            __("ARO specified (and its children, if any) will have ALLOW access "),
            __("to the specified ACO action (and the ACO's children, if any).")
        ],
        'arguments' => [
            'aro' => ['help' => __('ARO to check.'), 'required' => true],
            'aco' => ['help' => __('ACO to check.'), 'required' => true],
            'action' => ['help' => __('Action to check')]
        ]
    ]);
}
```

Recebendo ajuda das Shells

Com a adição de ConsoleOptionParser receber ajuda de shells é feito de uma forma consistente e uniforme. Ao usar a opção `--help` ou `-h` você pode visualizar a ajuda para qualquer núcleo shell, e qualquer shell que implementa um ConsoleOptionParser:

```
cake bake --help
cake bake -h
```

Ambos devem gerar a ajuda para o bake. Se o shell suporta subcomandos você pode obter ajuda para estes de uma forma semelhante:

```
cake bake model --help
cake bake model -h
```

Isso deve fornecer a você a ajuda específica para a tarefa bake dos models.

Recebendo ajuda como XML

Quando a construção de ferramentas automatizadas ou ferramentas de desenvolvimento que necessitam interagir com shells do CakePHP, é bom ter ajuda disponível em uma máquina capaz interpretar formatos. O ConsoleOptionParser pode fornecer ajuda em xml, definindo um argumento adicional:

```
cake bake --help xml
cake bake -h xml
```

O trecho acima deve retornar um documento XML com a ajuda gerada, opções, argumentos e subcomando para o shell selecionado. Um documento XML de amostra seria algo como:

```
<?xml version="1.0"?>
<shell>
  <command>bake fixture</command>
  <description>Generate fixtures for use with the test suite. You can use
    `bake fixture all` to bake all fixtures.</description>
  <epilog>
    Omitting all arguments and options will enter into an interactive
    mode.
  </epilog>
```

```

<subcommands/>
<options>
  <option name="--help" short="-h" boolean="1">
    <default/>
    <choices/>
  </option>
  <option name="--verbose" short="-v" boolean="1">
    <default/>
    <choices/>
  </option>
  <option name="--quiet" short="-q" boolean="1">
    <default/>
    <choices/>
  </option>
  <option name="--count" short="-n" boolean="">
    <default>10</default>
    <choices/>
  </option>
  <option name="--connection" short="-c" boolean="">
    <default>default</default>
    <choices/>
  </option>
  <option name="--plugin" short="-p" boolean="">
    <default/>
    <choices/>
  </option>
  <option name="--records" short="-r" boolean="1">
    <default/>
    <choices/>
  </option>
</options>
<arguments>
  <argument name="name" help="Name of the fixture to bake.
    Can use Plugin.name to bake plugin fixtures." required="">
    <choices/>
  </argument>
</arguments>
</shell>

```

Roteamento em Shells / CLI

Na interface de linha de comando (CLI), especificamente suas shells e tarefas, `env('HTTP_HOST')` e outras variáveis de ambiente webbrowser específica, não estão definidas.

Se você gerar relatórios ou enviar e-mails que fazem uso de `Router::url()`, estes conterão a máquina padrão `http://localhost/` e resultando assim em URLs inválidas. Neste caso, você precisa especificar o domínio manualmente. Você pode fazer isso usando o valor de configuração `App.fullBaseUrl` no seu bootstrap ou na sua configuração, por exemplo.

Para enviar e-mails, você deve fornecer a classe `CakeEmail` com o host que você deseja enviar o e-mail:

```

$email = new CakeEmail();
$email->domain('www.example.org');

```

Iste afirma que os IDs de mensagens geradas são válidos e adequados para o domínio a partir do qual os e-mails são enviados.

Métodos enganchados

`Cake\Console\ConsoleOptionParser::initialize()`

Inicializa a Shell para atuar como construtor de subclasses e permite configuração de tarefas antes de desenvolver a execução.

`Cake\Console\ConsoleOptionParser::startup()`

Inicia-se a Shell e exibe a mensagem de boas-vindas. Permite a verificação e configuração antes de comandar ou da execução principal.

Substitua este método se você quiser remover as informações de boas-vindas, ou outra forma modificar o fluxo de pré-comando.

Mais tópicos

Shell Helpers

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)¹¹⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Console Interativo (REPL)

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)¹¹⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Executando Shells como Cron Jobs

Uma coisa comum a fazer com um shell é torná-lo executado como um cronjob para limpar o banco de dados de vez em quando ou enviar newsletters. Isso é trivial para configurar, por exemplo:

```
* /5 * * * * cd /full/path/to/root && bin/cake myshell myparam
# * * * * * comando para executar
# | | | | |
# | | | | |
# | | | | | \----- day of week (0 - 6) (0 a 6 são de domingo a sábado, ou
↪ use os nomes)
# | | | | | \----- mês (1 - 12)
```

¹¹⁸ <https://github.com/cakephp/docs>

¹¹⁹ <https://github.com/cakephp/docs>

```
# /      /      \----- dia do mês (1 - 31)
# /      \----- hora (0 - 23)
# \----- minuto (0 - 59)
```

Você pode ver mais informações aqui: <https://pt.wikipedia.org/wiki/Crontab>

Dica: Use `-q` (ou `-quiet`) para silenciar qualquer saída para cronjobs.

I18N Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹²⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Completion Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹²¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Plugin Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹²² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Routes Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

¹²⁰ <https://github.com/cakephp/docs>

¹²¹ <https://github.com/cakephp/docs>

¹²² <https://github.com/cakephp/docs>

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹²³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Upgrade Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹²⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Server Shell

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹²⁵ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Cache Shell

Para ajudá-lo a gerenciar melhor os dados armazenados em cache a partir de um ambiente CLI, um comando shell está disponível para limpar os dados em cache que seu aplicativo possui:

```
// Limpar uma configuração de cache
bin/cake cache clear <configname>

// Limpar todas as configurações de cache
bin/cake cache clear_all
```

Novo na versão 3.3.0: Os comandos shell do cache foram adicionados na versão 3.3.0

¹²³ <https://github.com/cakephp/docs>

¹²⁴ <https://github.com/cakephp/docs>

¹²⁵ <https://github.com/cakephp/docs>

Depuração

Depuração é uma etapa inevitável e importante de qualquer ciclo de desenvolvimento. Ainda que o CakePHP não forneça nenhuma ferramenta que se conecte com qualquer IDE ou editor de texto, este oferece várias ferramentas que auxiliam na depuração e exibição de tudo que está sendo executado “por baixo dos panos” na sua aplicação.

Depuração Básica

debug (*mixed \$var, boolean \$showHtml = null, \$showFrom = true*)

A função `debug()` é uma função de escopo global que funciona de maneira similar a função PHP `print_r()`. A função `debug()` exibe os conteúdos de uma variável de diversas maneiras. Primeiramente, se você deseja exibir os dados no formato HTML, defina o segundo parâmetro como `true`. A função também exibe a linha e o arquivo de onde a mesma foi chamada.

A saída da função somente é exibida caso a variável `$debug` do core esteja definida com o valor `true`.

stackTrace()

A função `stackTrace()` é uma função de escopo global, função esta que permite que seja exibida a pilha de execução onde quer que a mesma tenha sido chamada.

breakpoint()

Novo na versão 3.1.

Se você tem o **Psysh**¹²⁶ instalado poderá usar esta função em ambientes de interface de linha de comando (CLI) para abrir um console interativo com o escopo local atual:

```
// Some code  
eval(breakpoint());
```

Abrirá um console interativo que poderá ser utilizado para avaliar variáveis locais e executar outros trechos de código. Você pode sair do depurador interativo executando os comandos `quit` ou `q` na sessão.

¹²⁶ <http://psysh.org/>

Usando a Classe Debugger

```
class Cake\Error\Debugger
```

Para usar o depurador, assegure que `Configure::read('debug')` esteja definida como `true`.

Valores de saída

```
static Cake\Error\Debugger::dump($var, $depth = 3)
```

O método `dump` exibe o conteúdo da variável, incluindo todas as propriedades e métodos (caso existam) da variável fornecida no primeiro parâmetro:

```
$foo = [1, 2, 3];

Debugger::dump($foo);

// Saídas
array(
    1,
    2,
    3
)

// Objeto
$car = new Car();

Debugger::dump($car);

// Saídas
object(Car) {
    color => 'red'
    make => 'Toyota'
    model => 'Camry'
    mileage => (int)15000
}
```

Criando Logs com Pilha de Execução

```
static Cake\Error\Debugger::log($var, $level = 7, $depth = 3)
```

Cria um log detalhado da pilha de execução no momento em que a mesma foi invocada. O método `log()` exibe dados similares ao “`Debugger::dump()`”, mas no arquivo `debug.log` ao invés do buffer de saída principal. É válido ressaltar que o diretório **tmp** e seu conteúdo devem ter permissão de escrita para o servidor web a fim de que a função `log()` consiga executar corretamente.

Gerando Pilhas de Execução

```
static Cake\Error\Debugger::trace($options)
```

Retorna a pilha de execução atual. Cada linha inclui o método que chamou, qual arquivo e linha do qual a chamada foi originada:

```
// Em PostsController::index()
pr(Debugger::trace());

// Saídas
PostsController::index() - APP/Controller/DownloadsController.php, line 48
Dispatcher::_invoke() - CORE/src/Routing/Dispatcher.php, line 265
Dispatcher::dispatch() - CORE/src/Routing/Dispatcher.php, line 237
[main] - APP/webroot/index.php, line 84
```

Abaixo encontra-se a pilha de execução gerada ao chamar `Debugger::trace()` em uma ação de um controller. A leitura do fim para o início da pilha exibe a ordem de execução das funções.

Pegando Trechos de Arquivos

static `Cake\Error\Debugger::excerpt($file, $line, $context)`

Colete um trecho de um arquivo localizado em `$path` (caminho absoluto), na linha `$line` com número de linhas em torno deste trecho `$context`:

```
pr(Debugger::excerpt(ROOT . DS . LIBS . 'debugger.php', 321, 2));

// Gera como saída o seguinte:
Array
(
    [0] => <code><span style="color: #000000"> * @access public</span></code>
    [1] => <code><span style="color: #000000"> */</span></code>
    [2] => <code><span style="color: #000000">         function excerpt($file, $line,
    ↪$context = 2) {</span></code>

    [3] => <span class="code-highlight"><code><span style="color: #000000">
    ↪$data = $lines = array();</span></code></span>
    [4] => <code><span style="color: #000000">         $data = @explode("\n", file_get_
    ↪contents($file));</span></code>
)
```

Ainda que este método seja usado internamente, o mesmo pode ser conveniente caso você esteja criando suas próprias mensagens de erros e registros de logs.

static `Cake\Error\Debugger::getType($var)`

Obtém o tipo da variável. Caso seja um objeto, o retorno do método será o nome de sua classe

Usando Logging para Depuração

Registrar as mensagens é uma outra boa maneira de se depurar aplicações. Para isto, pode ser usada a classe `Cake\Log\Log` para fazer o logging na sua aplicação. Todos os objetos que fazem uso de `LogTrait` têm um método de instanciação `log()` que pode ser usado para registrar mensagens:

```
$this->log('Cheguei aqui', 'debug');
```

O código acima escreverá `Cheguei aqui` no arquivo de registros de depuração (debug log). Você pode usar seus registros para auxiliar na depuração de métodos que contêm redirecionamentos e laços complicados. Você poderá usar também `Cake\Log\Log::write()` para escrever mensagens nos registros. Esse método pode ser chamado de forma estática em qualquer lugar da sua aplicação, pressupondo-se que `Log` já esteja carregado:

```
// No início do arquivo que deseja registrar.  
use Cake\Log\Log;  
  
// Em qualquer lugar que Log tenha sido importado.  
Log::debug('Cheguei aqui');
```

Debug Kit

O DebugKit é um plugin composto por ótimas ferramentas de depuração. Uma dessas ferramentas é uma toolbar renderizada em HTML, na qual é possível visualizar uma grande quantidade de informações sobre sua aplicação e a atual requisição realizada pela mesma. Veja no capítulo *Debug Kit* como instalar e usar o DebugKit.

Implantação

Uma vez que sua aplicação está completa, ou mesmo antes quando você quiser colocá-la no ar. Existem algumas poucas coisas que você deve fazer quando colocar em produção uma aplicação CakePHP.

Atualizar config/app.php

Atualizar o arquivo **core.php**, especificamente o valor do `debug` é de extrema importância. Tornar o `debug` igual a `false` desabilita muitos recursos do processo de desenvolvimento que nunca devem ser expostos ao mundo. Desabilitar o `debug`, altera as seguintes coisas:

- Mensagens de depuração criadas com `pr()` e `debug()` serão desabilitadas.
- O cache interno do CakePHP será descartado após 999 dias ao invés de ser a cada 10 segundos como em desenvolvimento.
- Views de erros serão menos informativas, retornando mensagens de erros genéricas.
- Erros do PHP não serão mostrados.
- O rastreamento de stack traces (conjunto de exceções) será desabilitado.

Além dos itens citados acima, muitos plugins e extensões usam o valor do `debug` para modificarem seus comportamentos.

Por exemplo, você pode setar uma variável de ambiente em sua configuração do Apache:

```
SetEnv CAKEPHP_DEBUG 1
```

E então você pode definir o level de debug dinamicamente no **config/app.php**:

```
$debug = (bool) getenv('CAKEPHP_DEBUG');  
  
return [  
    'debug' => $debug,  
    .....  
];
```

Checar a segurança

Se você está jogando sua aplicação na selva, é uma boa idéia certificar-se que ela não possui vulnerabilidades óbvias:

- Certifique-se de utilizar o [Cross Site Request Forgery](#).
- Você pode querer habilitar o [Security](#). Isso pode prevenir diversos tipos de adulteração de formulários e reduzir a possibilidade de overdose de requisições.
- Certifique-se que seus models possuem as regras [Validação](#) de validação habilitadas.
- Verifique se apenas o seu diretório `webroot` é visível publicamente, e que seus segredos (como seu app salt, e qualquer chave de segurança) são privados e únicos também.

Definir a raiz do documento

Definir a raiz do documento da sua aplicação corretamente é um passo importante para manter seu código protegido e sua aplicação mais segura. As aplicações desenvolvidas com o CakePHP devem ter a raiz apontando para o diretório `webroot`. Isto torna a aplicação e os arquivos de configurações inacessíveis via URL. Configurar a raiz do documento depende de cada servidor web. Veja a [Reescrita de URL](#) para informações sobre servidores web específicos.

De qualquer forma você vai querer definir o host/domínio virtual para o `webroot/`. Isso remove a possibilidade de arquivos fora do diretório raiz serem executados.

Aprimorar a performance de sua aplicação

O carregamento de classes pode alocar facilmente o tempo de processamento de sua aplicação. A fim de evitar esse problema, é recomendado que você execute este comando em seu servidor de produção uma vez que a aplicação esteja implantada:

```
php composer.phar dumpautoload -o
```

Sabendo que manipulação de referências estáticas, como imagens, JavaScript e arquivos CSS, plugins, através do `Dispatcher` é incrivelmente ineficiente, é fortemente recomendado referenciá-los simbolicamente para produção. Por exemplo:

```
ln -s Plugin/YourPlugin/webroot/css/yourplugin.css webroot/css/yourplugin.css
```

Email

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹²⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Aviso: Antes da versão 3.1, as classes `Email` e `Transport` estavam com o namespace `Cake\Network\Email` em vez do namespace `Cake\Mailer`.

```
class Cake\Mailer\Email (mixed $profile = null)
```

`Email` é uma nova classe para enviar E-mail. Com essa classe você pode enviar e-mail de qualquer lugar em sua aplicação.

Uso Básico

Primeiro de tudo, você deve garantir que a classe está carregada:

```
use Cake\Mailer\Email;
```

Depois que você carregou `Email`, you pode enviar um e-mail com o seguinte:

```
$email = new Email('default');
$email->from(['remetente@example.com' => 'Meu Site'])
    ->to('destinatario@exemplo.com')
    ->subject('Assunto')
    ->send('Minha mensagem');
```

Com os métodos construtores da classe `Email`, você é capaz de definir suas propriedades com o encadeamento de método.

¹²⁷ <https://github.com/cakephp/docs>

Email tem vários métodos para definir os destinatários - `to()`, `cc()`, `bcc()`, `addTo()`, `addCc()` e `addBcc()`. A diferença é que os três primeiros irão substituir o que já foi definido antes e mais tarde será apenas como adicionar mais destinatários ao seu respectivo campo:

```
$email = new Email();
$email->to('to@example.com', 'To Example');
$email->addTo('to2@example.com', 'To2 Example');
// Os destinatários são: to@example.com and to2@example.com
$email->to('test@example.com', 'ToTest Example');
// O destinatário é: test@example.com
```

Escolhendo Remetente

Quando enviamos um e-mail em nome de outra pessoa, é uma boa ideia definirmos quem é o remetente original usando o cabeçalho Sender. Você pode fazer isso usando `sender()`:

```
$email = new Email();
$email->sender('app@example.com', 'MyApp emailer');
```

Nota: É também uma boa ideia para definir o envelope remetente quando enviar um correio em nome de outra pessoa. Isso as impede de obter quaisquer mensagens sobre a capacidade de entrega.

Configuração

A configuração de Email padrão é criada usando `config()` e `configTransport()`. Você deve colocar as predefinições de e-mail no arquivo **config/app.php**. O arquivo **config/app.default.php** é um exemplo deste arquivo. Não é necessário definir a configuração de e-mail em **config/app.php**. Email pode ser usado sem ele e usar os respectivos métodos para definir todas as configurações separadamente ou carregar uma variedade de configurações.

Ao definir perfis e transportes, você pode manter o código do aplicativo livre dos dados de configuração, e evitar a duplicação que faz manutenção e implantação mais difícil.

Para carregar uma configuração pré-definida, você pode usar o método `profile()` ou passá-lo para o construtor de Email:

```
$email = new Email();
$email->profile('default');

// Ou no Construtor
$email = new Email('default');
```

Em vez de passar uma string que corresponde a um nome de configuração predefinida, você também pode apenas carregar uma variedade de opções:

```
$email = new Email();
$email->profile(['from' => 'me@example.org', 'transport' => 'my_custom']);

// Ou no Construtor
$email = new Email(['from' => 'me@example.org', 'transport' => 'my_custom']);
```

Alterado na versão 3.1: O perfil `default` do e-mail é automaticamente setado quando uma instância *Email* é criada.

Configurando Transportes

static Cake\Mailer\Email::configTransport(\$key, \$config = null)

As mensagens de email são entregues por transportes. Diferentes transportes permitem o envio de mensagens via funções PHP mail do PHP servidores SMTP (ou não em todos, que é útil para depuração. Configurar transportes permite-lhe manter os dados de configuração fora de seu código do aplicativo e torna a implantação mais simples, como você pode simplesmente mudar os dados de configuração. Um exemplo de configuração de transporte é parecido com:

```
use Cake\Mailer\Email;

// Configuração Simples de Email
Email::configTransport('default', [
    'className' => 'Mail'
]);

// Configuração smtp Simples
Email::configTransport('gmail', [
    'host' => 'ssl://smtp.gmail.com',
    'port' => 465,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtplib'
]);
```

Você pode configurar servidores SSL SMTP, como o Gmail. Para fazer isso, colocar o prefixo `ssl://` no hospedeiro e configurar o valor de porta em conformidade. Você também pode ativar TLS SMTP usando o `tls` opção:

```
use Cake\Mailer\Email;

Email::configTransport('gmail', [
    'host' => 'smtp.gmail.com',
    'port' => 587,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtplib',
    'tls' => true
]);
```

A configuração acima possibilita uma comunicação TLS para mensagens de e-mail.

Aviso: Você vai precisar ter ativado o acesso para aplicações menos seguras em sua conta do Google para que isso funcione: [Permitindo aplicações menos seguras para acessar sua conta](#)¹²⁸.

¹²⁸ <https://support.google.com/accounts/answer/6010255>

Nota: Para usar SSL + SMTP, você precisará ter o SSL configurado no seu PHP.

As opções de configuração também pode ser fornecido como uma string *DSN*. Isso é útil quando se trabalha com variáveis de ambiente ou prestadores *PaaS*:

```
Email::configTransport('default', [
    'url' => 'smtp://my@gmail.com:secret@smtp.gmail.com:465?tls=true',
]);
```

Ao usar uma string DSN você pode definir quaisquer parâmetros/opções adicionais como argumentos de string de consulta.

static `Cake\Mailer\Email::dropTransport($key)`

Uma vez configurado, os transportes não pode ser modificado. A fim de modificar um transporte, você deve primeiro soltá-lo e, em seguida, configurá-lo.

Perfis de Configuração

Definição de perfis de entrega permitem consolidar as configurações de e-mail comuns em perfis reutilizáveis. Seu aplicativo pode ter tantos perfis como necessário. As seguintes chaves de configuração são usados:

- `'from'`: E-mail ou array do remetente. Visto `Email::from()`.
- `'sender'`: E-mail ou array do Remetente original. Visto `Email::sender()`.
- `'to'`: E-mail ou array do Destinatário. Visto `Email::to()`.
- `'cc'`: E-mail ou array da Cópia de Carbono. Visto `Email::cc()`.
- `'bcc'`: E-mail ou array da cópia oculta. Visto `Email::bcc()`.
- `'replyTo'`: Email ou array do E-mail de respostas. Visto `Email::replyTo()`.
- `'readReceipt'`: Endereço de E-mail ou array de endereços para receber a recepção de leitura. Visto `Email::readReceipt()`.
- `'returnPath'`: Endereço de E-mail ou um array de endereços para retornar se teve alguns erros. Visto `Email::returnPath()`.
- `'messageId'`: ID da mensagem do e-mail. Visto `Email::messageId()`.
- `'subject'`: Assunto da mensagem. Visto `Email::subject()`.
- `'message'`: Conteúdo de mensagem. Não defina este campo se você estiver usando o conteúdo processado.
- `'headers'`: Cabeçalhos sejam incluídas. Visto `Email::setHeaders()`.
- `'viewRender'`: Se você estiver usando conteúdo renderizado, definir o nome da classe da view. Visto `Email::viewRender()`.
- `'template'`: Se você estiver usando conteúdo renderizado, definir o nome do template. Visto `Email::template()`.
- `'theme'`: Tema usado quando o template é renderizado. Visto `Email::theme()`.
- `'layout'`: Se você estiver usando conteúdo renderizado, definir o layout para renderizar. Se você quer renderizar um template sem layout, definir este campo como null. Visto `Email::template()`.
- `'viewVars'`: Se você estiver usando conteúdo renderizado, definir o array com as variáveis para serem usadas na view. Visto `Email::viewVars()`.
- `'attachments'`: Lista de arquivos para anexar. Visto `Email::attachments()`.
- `'emailFormat'`: Formato do e-mail (html, text ou both). Visto `Email::emailFormat()`.
- `'transport'`: Nome da configuração de transporte. Visto `Mailer\Email::configTransport()`.
- `'log'`: Nível de log para registrar os cabeçalhos de e-mail e mensagem. `true` usará `LOG_DEBUG`. Visto também como `CakeLog::write()`
- `'helpers'`: Array de helpers usado no template do e-mail.

Todas essas configurações são opcionais, exceto 'from'.

Nota: Os valores das chaves acima usando e-mail ou array, como from, to, cc, etc será passado como primeiro parâmetro de métodos correspondentes. O equivalente de: `Email::from('my@example.com', 'My Site')` pode ser definido como `'from' => ['my@example.com' => 'My Site']` na sua configuração.

Definindo Cabeçalho

Em `Email` você está livre para definir os cabeçalhos que você deseja. Quando migrar usando e-mail, não se esqueça de colocar o prefixo X- em seus cabeçalhos.

Visto como `Email::setHeaders()` e `Email::addHeaders()`.

Enviando E-mail com Templates

E-mails são frequentemente muito mais do que apenas uma simples mensagem de texto. A fim de facilitar, o CakePHP fornece uma maneira de enviar e-mails usando o CakePHP. Veja em [view layer](#).

Os templates para e-mails residir em uma pasta especial em sua aplicação no diretório `Template` chamado `Email`. Visualizações de e-mail também pode usar layouts e os elementos assim como vistas normais:

```
$email = new Email();
$email->template('welcome', 'fancy')
    ->emailFormat('html')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

O acima usaria `src/Template/Email/html/welcome.ctp` para a vista e `src/Template/Layout/E-mail/html/fancy.ctp` para o layout. Você pode enviar mensagens de e-mail com templates de várias partes, veja:

```
$email = new Email();
$email->template('welcome', 'fancy')
    ->emailFormat('both')
    ->to('bob@example.com')
    ->from('app@domain.com')
    ->send();
```

Este usaria os seguintes arquivos de template:

- `src/Template/Email/text/welcome.ctp`
- `src/Template/Layout/Email/text/fancy.ctp`
- `src/Template/Email/html/welcome.ctp`
- `src/Template/Layout/Email/html/fancy.ctp`

Ao enviar e-mails com templates, você tem a opção de enviar qualquer `text`, `html` ou `both`.

Você pode definir as variáveis da view com `Email::viewVars()`:

```
$email = new Email('templated');
$email->viewVars(['value' => 12345]);
```

Em seus templates de e-mail, você pode usar isso com:

```
<p>Aqui está o seu valor: <b><?= $value ?></b></p>
```

Você pode usar helpers em e-mails, bem como você pode em arquivos de modelo normais. Por padrão, somente o `HtmlHelper` é carregado. Você pode carregar helpers adicionais usando os métodos `helpers()`:

```
$email->helpers(['Html', 'Custom', 'Text']);
```

Ao definir ajudantes se esqueça de incluir 'Html' ou ele será removido do helpers carregado no seu template de e-mail.

Se você quiser enviar e-mail usando templates em um plugin, você pode usar o familiar *Sintaxe Plugin* para fazê-lo:

```
$email = new Email();  
$email->template('Blog.new_comment', 'Blog.auto_message');
```

O acima usaria templates a partir do plug-in Blog como um exemplo.

Em alguns casos, pode ser necessário substituir o template padrão fornecido pelo plugins. Você pode fazer isso usando temas, dizendo para o E-mail usar o tema apropriado usando o método `Email::theme()`:

```
$email = new Email();  
$email->template('Blog.new_comment', 'Blog.auto_message');  
$email->theme('TestTheme');
```

Isso permite que você substitua o `new_comment` em seu tema, sem modificar o plug-in Blog. O arquivo de template precisa ser criado no seguinte caminho: **src/Template/Plugin/TestTheme/Blog/Email/text/new_comment.ctp**.

Envio de Anexos

`Cake\Mailer\Email::attachments($attachments = null)`

Você pode anexar arquivos a mensagens de email também. Há alguns diferentes formatos, dependendo do tipo de arquivos que você tem, e como você quer os nomes dos arquivos para aparecer no email do destinatário:

1. String: `$email->attachments('/full/file/path/file.png')` irá anexar este arquivo com o nome `file.png`.
2. Array: `$email->attachments(['/full/file/path/file.png'])` tem o mesmo comportamento como o uso de uma String.
3. Array com chave: `$email->attachments(['photo.png' => '/full/some_hash.png'])` irá anexar alguns `hash.png` com o nome `photo.png`. O destinatário verá `photo.png`, não `hash.png`.
4. Arrays aninhados:

```
$email->attachments([  
    'photo.png' => [  
        'file' => '/full/some_hash.png',  
        'mimetype' => 'image/png',  
        'contentId' => 'my-unique-id'  
    ]  
]);
```

O acima irá anexar o arquivo com diferentes `mimetypes` e com identificação de conteúdo personalizado (quando definir o ID de conteúdo do anexo é transformado para linha). O `mimetype` e `contentId` são opcionais nessa forma.

- 4.1. Quando você estiver usando o `contentId`, você pode usar o arquivo no corpo HTML como ``.
- 4.2. Você pode usar a opção `contentDisposition` conteúdo para desativar cabeçalho `Content-Disposition` para um anexo. Isso é útil quando é feito o envio de convites para o iCal para clientes usando o Outlook.
- 4.3 Em vez de a opção `file` você pode fornecer o conteúdo do arquivo como uma string usando a opção `data`. Que lhe permite anexar arquivos sem a necessidade de caminhos de arquivo para eles.

Usando Transportes

Transportes são classes atribuídas a enviar o e-mail sobre algum protocolo ou método. CakePHP suporta o o transporte de Mail (padrão), Debug e SMTP.

Para configurar o método, você deve usar o método `Cake\Mailer\Email::transport()` ou ter o transporte em sua configuração:

```
$email = new Email();

// Usar um transporte chamado já configurado usando Email::configTransport()
$email->transport('gmail');

// Usando um método Construtor
$transport = new DebugTransport();
$email->transport($transport);
```

Erros & Exceções

Error & Exception Configuration

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹²⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹²⁹ <https://github.com/cakephp/docs>

Sistema de eventos

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹³⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹³⁰ <https://github.com/cakephp/docs>

Internacionalização e Localização

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](#)¹³¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Uma das melhores maneiras para uma aplicação alcançar uma maior audiência é atender a vários idiomas. Isso muitas vezes pode provar ser uma tarefa difícil, mas a internacionalização e recursos de localização do CakePHP tornam muito mais fácil.

Primeiro, é importante entender a terminologia. *Internacionalização* refere-se à capacidade de um aplicativo ser localizado. O termo *localização* refere-se à adaptação de uma aplicação, para atender idioma específico (ou cultura) requisitos (Isto é, uma “localidade”). Internacionalização e Localização são frequentemente abreviado como *i18n* (internationalization) e *l10n* (localization); 18 e 10 são o número de caracteres entre a primeira e última letra de cada termo.

Configurando Traduções

Existem apenas alguns passos para ir de um aplicativo de um único idioma a uma aplicação multi-lingual, o primeiro deles é fazer uso da função `:php:func: __()` em seu código. Abaixo está um exemplo de algum código para uma aplicação de um único idioma:

```
<h2>Popular Articles</h2>
```

Para Internacionalizar seu código, tudo que você precisa fazer é refatorar a string usando `:php:func: __()` por exemplo:

```
<h2><?= __('Popular Articles') ?></h2>
```

Fazendo nada mais, estes dois exemplos de código são funcionalmente idênticos - ambos irão enviar o mesmo conteúdo para o navegador. A função `:php:func: __()` irá traduzir a string passada se a tradução estiver disponível, ou devolvê-la inalterada.

¹³¹ <https://github.com/cakephp/docs>

Arquivos de Idiomas

Traduções podem ser disponibilizados usando arquivos de idiomas armazenados na aplicação. O formato padrão para arquivos de tradução do CakePHP é o formato [Gettext](#)¹³². Os arquivos precisam ser colocado dentro do Diretório **src/Locale/** e dentro deste diretório, deve haver uma subpasta para cada idioma, por exemplo:

```
/src
  /Locale
    /en_US
      default.po
    /en_GB
      default.po
      validation.po
    /es
      default.po
```

O domínio padrão é 'default', portanto, a pasta **src/Locale/** deve pelo menos conter o arquivo **default.po** como mostrado acima. Um domínio refere-se a qualquer arbitrário agrupamento de mensagens de tradução. Quando nenhum grupo é usado, o grupo padrão é selecionado.

As mensagens das Strings do core extraídos da biblioteca CakePHP podem ser armazenado separadamente em um arquivo chamado **cake.po** em **src/Locale/**. O [CakePHP localized library](#)¹³³ possui traduções para as mensagens traduzidas voltados para o cliente no núcleo (o domínio Cake). Para usar esses arquivos, baixar ou copiá-los para o seu local esperado: **src/Locale/<locale>/cake.po**. Se sua localidade está incompleta ou incorreta, por favor envie um PR neste repositório para corrigi-lo.

Plugins também podem conter arquivos de tradução, a convenção é usar o `under_score` do nome do plugin como o domínio para a tradução mensagens:

```
MyPlugin
  /src
    /Locale
      /fr
        my_plugin.po
      /de
        my_plugin.po
```

Pastas de tradução pode ser o código ISO de duas letras do idioma ou nome do local completo, como `fr_FR`, `es_AR`, `da_DK` que contém tanto o idioma e o país onde ele é falado.

Um exemplo de arquivo de tradução pode ser visto como:

```
msgid "My name is {0}"
msgstr "Je m'appelle {0}"

msgid "I'm {0,number} years old"
msgstr "J'ai {0,number} ans"
```

Extraindo arquivos .pot com I18n Shell

Para criar os arquivos .pot a partir de `__()` e outros tipos de mensagens internacionalizadas que podem ser encontrados no código do aplicativo, você pode usar o shell `i18n`. Por favor, leia o [Capítulo Seguinte](#) para saber mais.

¹³² <http://en.wikipedia.org/wiki/Gettext>

¹³³ <https://github.com/cakephp/localized>

Definir a localidade padrão

A localidade padrão pode ser definida em no arquivo **config/app.php**, definindo `App::defaultLocale`:

```
'App' => [
    ...
    'defaultLocale' => env('APP_DEFAULT_LOCALE', 'en_US'),
    ...
]
```

Isto vai controlar vários aspectos da aplicação, incluindo o padrão da linguagem de traduções, o formato da data, formato de número e moeda sempre que qualquer daqueles é exibida usando as bibliotecas de localização que o CakePHP fornece.

Alterando o local em tempo de execução

Para alterar o idioma para as mensagens traduzidas você pode chamar esse método:

```
use Cake\I18n\I18n;

I18n::locale('de_DE');
```

Isso também irá alterar a forma como números e datas são formatadas quando usamos uma das ferramentas de localização.

Usando funções de tradução

CakePHP fornece várias funções que o ajudarão a internacionalizar sua aplicação. O mais utilizado é `:php:func: __()`. Esta função é usada para recuperar uma única mensagem de tradução ou devolver a mesma String se não houver tradução:

```
echo __('Popular Articles');
```

Se você precisa agrupar suas mensagens, por exemplo, traduções dentro de um plugin, você pode usar a função `:php:func: __d()` para buscar mensagens de outro domínio:

```
echo __d('my_plugin', 'Trending right now');
```

Às vezes traduções de Strings podem ser ambíguos para as pessoas traduzindo-os. Isso pode acontecer se duas sequências são idênticas, mas referem-se a coisas diferentes. Por exemplo, “letter” tem vários significados em Inglês. Para resolver esse problema, você pode usar a função `:php:func: __x()`:

```
echo __x('written communication', 'He read the first letter');

echo __x('alphabet learning', 'He read the first letter');
```

O primeiro argumento é o contexto da mensagem e a segunda é a mensagem a ser traduzida.

Usando variáveis em mensagens de tradução

Funções de tradução permitem que você interpole variáveis para as mensagens usando marcadores especiais definidos na própria mensagem ou na string traduzida:

```
echo __("Hello, my name is {0}, I'm {1} years old", ['Jefferson', 19]);
```

Marcadores são numéricos, e correspondem às teclas na matriz passada. Você pode também passar variáveis como argumentos independentes para a função:

```
echo __("Small step for {0}, Big leap for {1}", 'Man', 'Humanity');
```

Todas as funções de tradução apoiam as substituições de espaço reservado:

```
__d('validation', 'The field {0} cannot be left empty', 'Name');

__x('alphabet', 'He read the letter {0}', 'Z');
```

O caractere ' (aspas simples) age como um código de escape na mensagem de tradução. Todas as variáveis entre aspas simples não serão substituídos e é tratado como texto literal. Por exemplo:

```
__("This variable '{0}' be replaced.", 'will not');
```

Ao usar duas aspas adjacentes suas variáveis e serão substituídos adequadamente:

```
__("This variable ''{0}'' be replaced.", 'will');
```

Estas funções tiram vantagem do [UTI MessageFormatter](#)¹³⁴ para que possa traduzir mensagens e localizar datas, números e moeda, ao mesmo tempo:

```
echo __(
    'Hi {0,string}, your balance on the {1,date} is {2,number,currency}',
    ['Charles', '2014-01-13 11:12:00', 1354.37]
);

// Returns
Hi Charles, your balance on the Jan 13, 2014, 11:12 AM is $ 1,354.37
```

Os números em espaços reservados podem ser formatados, bem como com o controle de grão fino da saída:

```
echo __(
    'You have traveled {0,number,decimal} kilometers in {1,number,integer} weeks',
    [5423.344, 5.1]
);

// Returns
You have traveled 5,423.34 kilometers in 5 weeks

echo __('There are {0,number,#,###} people on earth', 6.1 * pow(10, 8));

// Returns
There are 6,100,000,000 people on earth
```

Esta é a lista de especificadores de formato que você pode colocar após a palavra `number`:

- `integer`: Remove a parte Decimal
- `decimal`: Formata o número como um float
- `currency`: Coloca o local do símbolo de moeda e números de casas decimais
- `percent`: Formata o número como porcentagem

¹³⁴ <http://php.net/manual/en/messageformatter.format.php>

Datas também pode ser formatadas usando a palavra `date` após o número do espaço reservado. Uma lista de opções adicionais a seguir:

- `short`
- `medium`
- `long`
- `full`

A palavra `time` após o número de espaço reservado também é aceito e compreende as mesmas opções que `date`.

Nota: Espaços reservados nomeados são suportados no PHP 5.5+ e são formatados como `{name}`. Ao usar espaços reservados nomeados para passar as variáveis em uma matriz usando pares de chave/valor, por exemplo `['name' => 'Jefferson', 'age' => 19]`.

Recomenda-se usar o PHP 5.5 ou superior ao fazer uso de recursos de internacionalização no CakePHP. A extensão `php5-intl` deve ser instalada e a versão UTI deve estar acima 48.x.y (para verificar a versão UTI `Intl::getIcuVersion ()`).

Plurais

Uma parte crucial de internacionalizar sua aplicação é a pluralização das suas mensagens corretamente, dependendo do idioma que eles são mostrados. O CakePHP fornece algumas maneiras de selecionar corretamente plurais em suas mensagens.

Usando UTI para Seleção de Plural

O primeiro está aproveitando o formato de mensagem ICU que vem por padrão nas funções de tradução. Nos arquivos de traduções você pode ter as seguintes cadeias

```
msgid "{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}"
msgstr "{0,plural,=0{Nenhum resultado} =1{1 resultado} other{# resultados}}"

msgid "{placeholder,plural,=0{No records found} =1{Found 1 record} other{Found {1}
↳records}}"
msgstr "{placeholder,plural,=0{Nenhum resultado} =1{1 resultado} other{{1} resultados
↳}"
```

E na aplicação utilize o seguinte código para a saída de uma das traduções para essa sequência:

```
__('{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}', [0]);

// Returns "Ningún resultado" as the argument {0} is 0

__('{0,plural,=0{No records found} =1{Found 1 record} other{Found # records}}', [1]);

// Returns "1 resultado" because the argument {0} is 1

__('{placeholder,plural,=0{No records found} =1{Found 1 record} other{Found {1}
↳records}}', [0, 'many', 'placeholder' => 2])

// Returns "many resultados" because the argument {placeholder} is 2 and
// argument {1} is 'many'
```

Um olhar mais atento para o formato que acabamos utilizado tornará evidente como as mensagens são construídas:

```
{ [count placeholder],plural, case1{message} case2{message} case3{...} ... }
```

O [count placeholder] pode ser o número-chave de qualquer das variáveis que você passar para a função de tradução. Ele será usado para selecionar o plural correto.

Note que essa referência para [count placeholder] dentro de {message} você tem que usar #.

Você pode usar ids de mensagem mais simples se você não deseja digitar a plena sequência de seleção para plural em seu código

```
msgid "search.results"
msgstr "{0,plural,=0{Nenhum resultado} =1{1 resultado} other{{1} resultados}}"
```

Em seguida, use a nova string em seu código:

```
__('search.results', [2, 2]);

// Returns: "2 resultados"
```

A última versão tem a desvantagem na qual existe uma necessidade de arquivar mensagens e precisa de tradução para o idioma padrão mesmo, mas tem a vantagem de que torna o código mais legível.

Às vezes, usando o número de correspondência direta nos plurais é impraticável. Por exemplo, idiomas como o árabe exigem um plural diferente quando você se refere a algumas coisas. Nesses casos, você pode usar o UTI correspondentes. Em vez de escrever:

```
=0{No results} =1{...} other{...}
```

Você pode fazer:

```
zero{No Results} one{One result} few{...} many{...} other{...}
```

Certifique-se de ler a [Language Plural Rules Guide](http://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html)¹³⁵ para obter uma visão completa dos aliases que você pode usar para cada idioma.

Usando Gettext para Seleção de Plural

O segundo formato para seleção de plural aceito é a utilização das capacidades embutidas de Gettext. Neste caso, plurais será armazenado nos arquivos .po, criando uma linha de tradução de mensagens separada por forma de plural:

```
# One message identifier for singular
msgid "One file removed"
# Another one for plural
msgid_plural "{0} files removed"
# Translation in singular
msgstr[0] "Un fichero eliminado"
# Translation in plural
msgstr[1] "{0} ficheros eliminados"
```

Ao usar este outro formato, você é obrigado a usar outra tradução de forma funcional:

```
// Returns: "10 ficheros eliminados"
$count = 10;
__n('One file removed', '{0} files removed', $count, $count);
```

¹³⁵ http://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html


```
// Também é possível utilizá-lo dentro de um domínio
__dn('my_plugin', 'One file removed', '{0} files removed', $count, $count);
```

O número dentro de `msgstr[]` é o número atribuído pela Gettext para o plural na forma da língua. Algumas línguas têm mais de duas formas plurais, para exemplo *Croatian*:

```
msgid "One file removed"
msgid_plural "{0} files removed"
msgstr[0] "{0} datoteka je uklonjena"
msgstr[1] "{0} datoteke su uklonjene"
msgstr[2] "{0} datoteka je uklonjeno"
```

Por favor visite a [Launchpad languages page](https://translations.launchpad.net/+languages)¹³⁶ para uma explicação detalhada dos números de formulário de plurais para cada idioma.

Criar seus próprios Tradutores

Se você precisar a divergir convenções do CakePHP sobre onde e como as mensagens de tradução são armazenadas, você pode criar seu próprio carregador de mensagem de tradução. A maneira mais fácil de criar o seu próprio tradutor é através da definição de um carregador para um único domínio e localidade:

```
use Aura\Intl\Package;

I18n::translator('animals', 'fr_FR', function () {
    $package = new Package(
        'default', // The formatting strategy (ICU)
        'default' // The fallback domain
    );
    $package->setMessages([
        'Dog' => 'Chien',
        'Cat' => 'Chat',
        'Bird' => 'Oiseau'
        ...
    ]);

    return $package;
});
```

O código acima pode ser adicionado ao seu **config/bootstrap.php** de modo que as traduções podem ser encontradas antes de qualquer função de tradução é usada. O mínimo absoluto que é necessário para a criação de um tradutor é que a função do carregador deve retornar um `Aura\Intl\Package` objeto. Uma vez que o código é no lugar que você pode usar as funções de tradução, como de costume:

```
I18n::locale('fr_FR');
__d('animals', 'Dog'); // Retorna "Chien"
```

Como você vê objetos, `Package` carregam mensagens de tradução como uma matriz. Você pode passar o método `setMessages()` da maneira que quiser: com código inline, incluindo outro arquivo, chamar outra função, etc. CakePHP fornece algumas funções da carregadeira que podem ser reutilizadas se você só precisa mudar para onde as mensagens são carregadas. Por exemplo, você ainda pode usar **.po**, mas carregado de outro local:

¹³⁶ <https://translations.launchpad.net/+languages>

```
use Cake\I18n\MessagesFileLoader as Loader;

// Load messages from src/Locale/folder/sub_folder/filename.po

I18n::translator(
    'animals',
    'fr_FR',
    new Loader('filename', 'folder/sub_folder', 'po')
);
```

Logging

Logging Configuration

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)¹³⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Formulários sem Models #####m

class Cake\Form\Form

Muitas vezes você precisará ter formulários associados ao *ORM entities* e *ORM tables* ou outras persistência de dados, mas há vezes quando você precisará validar um campo de usuário e então realizar uma ação se o dado é válido. O exemplo mais comum para esta situação é o formulário de contato.

¹³⁷ <https://github.com/cakephp/docs>

Criando o Formulário

Geralmente quando se usa a classe `Form` será necessário utilizar uma sub-classe para definir seu formulário. Isso facilita o teste, e permite o reuso do formulário. Formulários ficam dentro de **src/Form** e comumente tem `Form` como sufixo da classe. Por exemplo, um simples formulário de contato poderia ficar assim:

```
// em src/Form/ContactForm.php
namespace App\Form;

use Cake\Form\Form;
use Cake\Form\Schema;
use Cake\Validation\Validator;

class ContactForm extends Form
{
    protected function _buildSchema(Schema $schema)
    {
        return $schema->addField('name', 'string')
            ->addField('email', ['type' => 'string'])
            ->addField('body', ['type' => 'text']);
    }

    protected function _buildValidator(Validator $validator)
    {
        return $validator->add('name', 'length', [
            'rule' => ['minLength', 10],
            'message' => 'A name is required'
        ]->add('email', 'format', [
            'rule' => 'email',
            'message' => 'A valid email address is required',
        ]);
    }

    protected function _execute(array $data)
    {
        // Envie um email.
        return true;
    }
}
```

No exemplo acima vemos os 3 métodos hooks que o formulário fornece:

- `_buildSchema` é usado para definir o esquema de dados usado pelo `FormHelper` para criar um formulário HTML. Você pode definir o tipo de campo, tamanho, e precisão.
- `_buildValidator` Pega uma instância do `Cake\Validation\Validator` que você pode juntar com os validadores.
- `_execute` permite definir o comportamento desejado quando o `execute()` é chamado e o dado é válido.

Você sempre pode adicionar métodos públicos a sua maneira.

Processando Requisição de Dados

Uma vez definido o formulário, é possível usá-lo em seu controller para processar e validar os dados:

```
// No Controller
namespace App\Controller;

use App\Controller\AppController;
use App\Form\ContactForm;

class ContactController extends AppController
{
    public function index()
    {
        $contact = new ContactForm();
        if ($this->request->is('post')) {
            if ($contact->execute($this->request->getData())) {
                $this->Flash->success('We will get back to you soon.');
            } else {
                $this->Flash->error('There was a problem submitting your form.');
            }
        }
        $this->set('contact', $contact);
    }
}
```

No exemplo acima, usamos o método `execute()` para chamar o nosso método `_execute()` do formulário apenas quando o dado é válido, e definimos as mensagens flash adequadas. Poderíamos também ter usado o método `validate()` apenas para validar a requisição de dados:

```
$isValid = $form->validate($this->request->getData());
```

Definindo os Valores do Formulário

Na sequência para definir os valores para os campos do formulário sem modelo, basta apenas definir os valores usando `$this->request->getData()`, como em todos os outros formulários criados pelo `FormHelper`:

```
// Em um controller
namespace App\Controller;

use App\Controller\AppController;
use App\Form\ContactForm;

class ContactController extends AppController
{
    public function index()
    {
        $contact = new ContactForm();
        if ($this->request->is('post')) {
            if ($contact->execute($this->request->getData())) {
                $this->Flash->success('Retornaremos o contato em breve.');
            } else {
                $this->Flash->error('Houve um problema ao enviar seu formulário.');
            }
        }

        if ($this->request->is('get')) {
            //Values from the User Model e.g.
            $this->request->getData('name', 'John Doe');
            $this->request->getData('email', 'john.doe@example.com');
        }

        $this->set('contact', $contact);
    }
}
```

Valores devem apenas serem definidos se a requisição é do tipo GET, caso contrário você sobrescreverá os dados anteriormente passados via POST que de certa forma poderiam estar incorretos e não salvos.

Pegando os Erros do Formulário

Uma vez sido validado, o formulário pode recuperar seus próprios erros:

```
$errors = $form->errors();  
/* $errors contains  
[  
    'email' => ['A valid email address is required']  
]  
*/
```

Invalidando Campos Individuais do Formulário no Controller

É possível invalidar campos únicos do controller sem o uso da classe Validator. O Uso mais comum neste caso é quando a validação é feita no servidor remoto. Neste caso, você deve manualmente invalidar os campos de acordo com a resposta do servidor:

```
// em src/Form/ContactForm.php
public function setErrors($errors)
{
    $this->_errors = $errors;
}
```

Conforme como a classe validadora poderia ter retornado os erros, `$errors` deve estar neste formato:

```
["fieldName" => ["validatorName" => "The error message to display"]]
```

Agora você pode invalidar os campos determinar o `fieldName`, e então definir as mensagens de erro:

```
// Em um controller
$contact = new ContactForm();
$contact->setErrors(["email" => ["_required" => "Seu email é necessário"]]);
```

Prossiga para Criação do HTML com o FormHelper para ver o resultado.

Criando o HTML com FormHelper

Uma vez sido criado uma class Form, Once you've created a Form class, você provavelmente vai querer criar um formulário HTML para isso. FormHelper compreende objetos Form apenas como entidades ORM:

```
echo $this->Form->create($contact);  
echo $this->Form->input('name');  
echo $this->Form->input('email');  
echo $this->Form->input('body');  
echo $this->Form->button('Submit');  
echo $this->Form->end();
```

O código acima criar um formulário HTML para o `ContactForm` definidos anteriormente. Formulários HTML criados com FormHelper usará o esquema definido e validador para determinar os tipos de campos, tamanhos máximos, e validação de erros.

Plugins

O CakePHP permite que você configure uma combinação de controllers, models, e views, plugin de aplicativo empacotado que outros podem usar em suas aplicações CakePHP. Se você criou um módulo de gerenciamento de usuários, blog ou serviços da Web em uma das suas aplicações, por que não torna-lo um plugin CakePHP? Desta forma, você pode reutilizá-lo em seus outros aplicativos e compartilhar com a comunidade!

Um plugin do CakePHP é, em última instância, separado do próprio aplicativo host e, geralmente, oferece algumas funcionalidades bem definidas que podem ser embaladas de maneira ordenada e reutilizadas com pouco esforço em outras aplicações. O aplicativo e o plugin operam em seus respectivos espaços, mas compartilham propriedades específicas da aplicação (parâmetros de conectividade de banco de dados) que são definidos e compartilhados através da configuração do aplicativo.

No CakePHP 3.0 cada plugin define seu próprio namespace de nível superior. Por exemplo: `DebugKit`. Por convenção, os plugins usam o nome do pacote como seu namespace. Se você quiser usar um espaço para nome diferente, você pode configurar o espaço para nome do plugin, quando os plugins são carregados.

Instalando um Plugin com Composer

Muitos plugins estão disponíveis no *Packagist* <<http://packagist.org>> _ E podem ser instalado com *Composer*. Para instalar o `DebugKit`, você deve fazer assim o assim:

```
php composer.phar require cakephp/debug_kit
```

Isso instalará a versão mais recente do `DebugKit` e atualizaria seus arquivos **composer.json**, **composer.lock**, atualização **vendor/cakephp-plugins.php** e atualize seu autoloader.

Se o plugin que deseja instalar não estiver disponível em *Packagist.org*, você pode clonar ou copiar o código do plugin para seu diretório **plugins**. Supondo que você deseja instalar um plugin chamado ‘`ContactManager`’, você Deve ter uma pasta em **plugins** chamado ‘`ContactManager`’. Neste diretório São o src do plugin, testes e outros diretórios.

Plugin Map File

Ao instalar plugins através do *Composer*, você pode notar que **vendor/cakephp-plugins.php** é criado. Este arquivo de configuração contém um mapa de nomes de plugins e seus caminhos no sistema de arquivos. Isso torna possível para que os plugins sejam instalados no diretório padrão do vendor que está fora dos caminhos de pesquisa normais. A classe `Plugin` usará este arquivo para localizar plugins quando são carregados com `load ()` ou `loadAll`

() . Você geralmente não precisará editar este arquivo à mão, como Composer e plugin-installer O pacote o gerenciara para você.

Carregando um Plugin

Depois de instalar um plugin e configurar o autoloader, você deve carregar O plugin. Você pode carregar plugins um a um, ou todos eles com um único método:

```
// In config/bootstrap.php
// Or in Application::bootstrap()

// Carrega um único plugin
Plugin::load('ContactManager');

// Carrega um plugin com um namespace no nível superior.
Plugin::load('AcmeCorp/ContactManager');

// Carrega todos os plugins de uma só vez
Plugin::loadAll();
```

loadAll() carrega todos os plugins disponíveis, permitindo que você especifique determinadas configurações para plugins. load() funciona de forma semelhante, mas apenas carrega o Plugins que você especifica explicitamente.

Nota:

Plugin::loadAll() não irá carregar os plugins vendor namespaced que não são Definido em vendor/cakephp-plugins.php.

Há também um comando de shell acessível para habilitar o plugin. Execute a seguinte linha:

```
bin/cake plugin load ContactManager
```

Isso colocará o plugin Plugin::load('ContactManager'); no bootstrap para você.

Autoloading Plugin Classes

Ao usar bake para criar um plugin ou quando instalar um plugin usando o Composer, você normalmente não precisa fazer alterações em seu aplicativo para faça com que o CakePHP reconheça as classes que vivem dentro dele.

Em qualquer outro caso, você precisará modificar o arquivo do composer.json do seu aplicativo. Para conter as seguintes informações:

```
"psr-4": {
    (...)
    "MyPlugin\\": "./plugins/MyPlugin/src",
    "MyPlugin\\Test\\": "./plugins/MyPlugin/tests"
}
```

Se você estiver usando o vendor namespaces para seus plugins, o espaço para nome para mapeamento de caminho deve se parecer com o seguinte:

```
"psr-4": {
    (...)
    "AcmeCorp\\Users\\": "./plugins/AcmeCorp/Users/src",
```

```
"AcmeCorp\\Users\\Test\\": "./plugins/AcmeCorp/Users/tests"
}
```

Além disso, você precisará dizer ao Composer para atualizar o cache de autoloading:

```
$ php composer.phar dumpautoload
```

Se você não conseguir usar o Composer por qualquer outro motivo, você também pode usar um recurso alternativo Autoloading para o seu plugin:

```
Plugin::load('ContactManager', ['autoload' => true]);
```

Configuração do Plugin

Os métodos `load()` e `loadAll()` podem ajudar na configuração do plugin E roteamento. Talvez você queira carregar todos os plugins automaticamente enquanto especifica Rotas personalizadas e arquivos bootstrap para determinados plugins:

```
// No config/bootstrap.php,
// ou in Application::bootstrap()

// Usando loadAll()
Plugin::loadAll([
    'Blog' => ['routes' => true],
    'ContactManager' => ['bootstrap' => true],
    'WebmasterTools' => ['bootstrap' => true, 'routes' => true],
]);
```

Ou você pode carregar os plugins individualmente:

```
// Carregando apenas o blog e inclui rotas
Plugin::load('Blog', ['routes' => true]);

// Inclua o arquivo configuration/initializer do bootstrap.
Plugin::load('ContactManager', ['bootstrap' => true]);
```

Com qualquer uma das abordagens, você não precisa mais manualmente `include()` ou `require()` configuração de um plugin ou arquivo de rotas - acontece Automaticamente no momento e no lugar certos.

Você pode especificar um conjunto de padrões para `loadAll()` que irá aplicar a cada plugin que não tenha uma configuração mais específica.

O seguinte exemplo irá carregar o arquivo bootstrap de todos os plugins e além disso, as rotas do Blog Plugin:

```
Plugin::loadAll([
    ['bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

Tenha em atenção que todos os arquivos especificados deveriam existir na configuração o(s) plugin(s) ou PHP dará avisos para cada arquivo que não pode carregar. Você pode evitar potenciais avisos usando a opção `ignoreMissing`:

```
Plugin::loadAll([
    ['ignoreMissing' => true, 'bootstrap' => true],
    'Blog' => ['routes' => true]
]);
```

Ao carregar os plugins, o nome do plugin usado deve corresponder ao namespace. Para por exemplo, se você tiver um plugin com o namespace de nível superior `Users` você carregaria Usando:

```
Plugin::load('User');
```

Se você preferir ter seu nome vendor como nível superior e ter um espaço para nome como `AcmeCorp/Users`, então você carregaria o plugin como:

```
Plugin::load('AcmeCorp/Users');
```

Isso garantirá que os nomes das classes sejam resolvidos corretamente ao usar *sintaxe plugin*.

A maioria dos plugins indicará o procedimento adequado para configurá-los e configurar até o banco de dados em sua documentação. Alguns plugins exigirão mais configuração do que outros.

Usando Plugins

Você pode fazer referência aos controllers, models, components, behaviors, e helpers, prefixando o nome do plugin antes

Por exemplo, vamos supor que você queria usar o plugin do `ContactManager` `ContactInfoHelper` para produzir algumas informações de contato legítimas em uma das suas opiniões. No seu controller, o `$helpers` array poderia ficar assim:

```
public $helpers = ['ContactManager.ContactInfo'];
```

Nota: Esse nome de classe separado por pontos é denominado *sintaxe plugin*.

Você poderia então acessar o `ContactInfoHelper` como qualquer outro helper em sua view, como:

```
echo $this->ContactInfo->address($contact);
```

Criando seus próprios complementos

Apenas como um exemplo, vamos começar a criar o `ContactManager` plugin referenciado acima. Para começar, vamos configurar o nosso plugin estrutura de diretório básico. Deve ser assim:

```
/src
/plugins
  /ContactManager
    /config
    /src
      /Controller
      /Component
      /Model
      /Table
      /Entity
      /Behavior
    /View
      /Helper
    /Template
```

```

        /Layout
    /tests
        /TestCase
        /Fixture
/webroot

```

Observe o nome da pasta do plugin, ‘**ContactManager**’. É importante Que esta pasta tem o mesmo nome que o plugin.

Dentro da pasta do plugin, você notará que se parece muito com um aplicativo CakePHP, e é basicamente isso. Você não precisa incluir qualquer uma das pastas que você não está usando, ou seja, pode remover o que não for usar. Alguns plugins podem apenas define um Component e um Behavior, e nesse caso eles podem completamente omitir o diretório ‘Template’.

Um plugin também pode ter basicamente qualquer um dos outros diretórios de seu aplicativo, como Config, Console, webroot, etc.

Criando um plugin usando bake

O processo de criação de plugins pode ser bastante simplificado usando o bake shell.

Nota: Use sempre o bake para gerar código, isso evitará muitas dores de cabeça.

Para criar um plugin com o bake, use o seguinte comando:

```
bin/cake bake plugin ContactManager
```

Agora você pode user o bake com as mesmas convenções que se aplicam ao resto do seu aplicativo. Por exemplo - baking controllers:

```
bin/cake bake controller --plugin ContactManager Contacts
```

Consulte o capítulo *Geração de código com bake* se você tiver problemas para usar a linha de comando. Certifique-se de voltar a gerar o seu autoloader uma vez que você criou seu plugin:

```
$ php composer.phar dumpautoload
```

Rotas para Plugin

Os plugins podem fornecer arquivos de rotas contendo suas rotas. Cada plugin pode conter um arquivo **config/routes.php**. Este arquivo de rotas pode ser carregado quando o complemento é adicionado ou no arquivo de rotas do aplicativo. Para criar as rotas de plugins do ContactManager, coloque o seguinte **plugins/ContactManager/config/routes.php**:

```

<?php
use Cake\Routing\Route\DashedRoute;
use Cake\Routing\Router;

Router::plugin(
    'ContactManager',
    ['path' => '/contact-manager'],
    function ($routes) {

```

```
$routes->get('/contacts', ['controller' => 'Contacts']);
$routes->get('/contacts/:id', ['controller' => 'Contacts', 'action' => 'view
↪']);
$routes->put('/contacts/:id', ['controller' => 'Contacts', 'action' => 'update
↪']);
    }
};
```

O código acima irá conectar as rotas padrão para o seu plugin. Você pode personalizar isso no arquivo com rotas mais específicas mais tarde:

```
Plugin::load('ContactManager', ['routes' => true]);
```

Você também pode carregar rotas de plugins na lista de rotas do seu aplicativo. Fazendo isso fornece mais controle sobre como as rotas do plugin são carregadas e permite que você envolva as rotas de plugin em escopos ou prefixos adicionais:

```
Router::scope('/', function ($routes) {
    // Connect other routes.
    $routes->scope('/backend', function ($routes) {
        $routes->loadPlugin('ContactManager');
    });
});
```

O código acima resultaria em URLs como `/backend/contact_manager/contacts`.

Novo na versão 3.5.0: `RouteBuilder::loadPlugin()` foi adicionado in 3.5.0

Plugin Controllers

Os Controllers para o nosso plug-in do `ContactManager` serão armazenados em **plugins/ContactManager/src/Controller/**. Como a principal coisa que vamos estar fazendo gerenciar contatos, precisaremos de um `ContactsController` para este plugin.

Então, colocamos nosso new `ContactsController` em **plugins/ContactManager/src/Controller** e parece ser assim:

```
// plugins/ContactManager/src/Controller/ContactsController.php
namespace ContactManager\Controller;

use ContactManager\Controller\AppController;

class ContactsController extends AppController
{
    public function index()
    {
        //...
    }
}
```

Também faça o `AppController` se você não possuir um já:

```
// plugins/ContactManager/src/Controller/AppController.php
namespace ContactManager\Controller;

use App\Controller\AppController as BaseController;
```

```
class AppController extends BaseController
{
}
```

Um AppController do plugin pode manter a lógica do controller comum a todos os controllers em um plugin, mas não é necessário se você não quiser usar um.

Se você deseja acessar o que temos chegado até agora, visite `/contact-manager/contacts`. Você deve obter um erro “Missing Model” porque ainda não temos um model de Contact definido.

Se o seu aplicativo incluir o roteamento padrão do CakePHP, você será capaz de acessar seus controllers de plugins usando URLs como:

```
// Acesse a rota de índice de um controller de plugin.
/contact-manager/contacts

// Qualquer ação em um controller de plug-in.
/contact-manager/contacts/view/1
```

Se o seu aplicativo definir prefixos de roteamento, o roteamento padrão do CakePHP também conecte rotas que usam o seguinte padrão:

```
/:prefix/:plugin/:controller
/:prefix/:plugin/:controller/:action
```

Consulte a seção em [Configuração do Plugin](#) para obter informações sobre como carregar arquivos de rota específicos do plugin.

Para os plugins que você não criou com bake, você também precisará editar o **composer.json** para adicionar seu plugin às classes de autoload, isso pode ser feito conforme a documentação [Autoloading Plugin Classes](#).

Plugin Models

Os models para o plugin são armazenados em **plugins/ContactManager/src/Model**. Nós já definimos um Contacts-Controller para este plugin, então vamos criar a tabela e a entidade para esse controlador:

```
// plugins/ContactManager/src/Model/Entity/Contact.php:
namespace ContactManager\Model\Entity;

use Cake\ORM\Entity;

class Contact extends Entity
{
}

// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
}
```

Se você precisa fazer referência a um modelo no seu plugin ao criar associações ou definindo classes de entidade, você precisa incluir o nome do plugin com a class name, separado com um ponto. Por exemplo:

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasMany('ContactManager.AltName');
    }
}
```

Se você preferir que as chaves da array para a associação não tenham o prefixo plugin sobre eles, use a sintaxe alternativa:

```
// plugins/ContactManager/src/Model/Table/ContactsTable.php:
namespace ContactManager\Model\Table;

use Cake\ORM\Table;

class ContactsTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasMany('AltName', [
            'className' => 'ContactManager.AltName',
        ]);
    }
}
```

Você pode usar TableRegistry para carregar suas tabelas de plugins usando o familiar *sintaxe plugin*:

```
use Cake\ORM\TableRegistry;

$contacts = TableRegistry::get('ContactManager.Contacts');
```

Alternativamente, a partir de um contexto de controller, você pode usar:

```
$this->loadModel('ContactsMangager.Contacts');
```

Plugin Views

As views se comportam exatamente como ocorrem em aplicações normais. Basta colocá-los na pasta `plugins/[PluginName]/src/Template/`. Para nós o plugin ContactManager, precisamos de uma view para o nosso `ContactsController::index()` action, então incluíamos isso também:

```
// plugins/ContactManager/src/Template/Contacts/index.ctp:
<h1>Contacts</h1>
<p>Following is a sortable list of your contacts</p>
<!-- A sortable list of contacts would go here....-->
```


Os plugins podem fornecer seus próprios layouts. Para adicionar layouts em plugins, coloque seus arquivos de template dentro `plugins/[PluginName]/src/Template/Layout`. Para usar um layout de plug-in em seu controller você pode fazer o seguinte:

```
public $layout = 'ContactManager.admin';
```

Se o prefixo do plugin for omitido, o arquivo layout/view será localizado normalmente.

Nota: Para obter informações sobre como usar elementos de um plugin, procure [Elements](#)

Substituindo Templates de plugins do na sua aplicação

Você pode substituir todas as view do plugin do seu aplicativo usando caminhos especiais. E se você tem um plugin chamado ‘ContactManager’, você pode substituir os arquivos do template do plugin com lógica de visualização específica da aplicação criando arquivos usando o seguinte template `src/Template/Plugin/[Plugin]/[Controller]/[view].ctp`. Para o controller Contacts você pode fazer o seguinte arquivo:

```
src/Template/Plugin/ContactManager/Contacts/index.ctp
```

Criar este arquivo permitiria que você substituir `plugins/ContactManager/src/Template/Contacts/index.ctp`.

Se o seu plugin estiver em uma dependência no composer (ou seja, ‘TheVendor/ThePlugin’), o caminho para da view ‘index’ do controller personalizado será:

```
src/Template/Plugin/TheVendor/ThePlugin/Custom/index.ctp
```

Criar este arquivo permitiria que você substituir `vendor/thevendor/theplugin/src/Template/Custom/index.ctp`.

Se o plugin implementar um prefixo de roteamento, você deve incluir o prefixo de roteamento em seu O template para substitui.

Se o plugin ‘Contact Manager’ implementou um prefixo ‘admin’, o caminho principal seria:

```
src/Template/Plugin/ContactManager/Admin/ContactManager/index.ctp
```

Plugin Assets

Os recursos da web de um plugin (mas não arquivos PHP) podem ser atendidos através do plugin no diretório webroot, assim como os assets da aplicação principal:

```
/plugins/ContactManager/webroot/  
    css/  
    js/  
    img/  
    flash/  
    pdf/
```

Você pode colocar qualquer tipo de arquivo em qualquer no diretório webroot.

Aviso: Manipulação de assets estáticos (como imagens, JavaScript e arquivos CSS) Através do Dispatcher é muito ineficiente. Ver [Aprimorar a performance de sua aplicação](#) Para maiores informações.

Linking to Assets in Plugins

Você pode usar o *sintaxe plugin* ao vincular aos recursos do plugin usando o `View\Helper\HtmlHelper` script, image ou css methods:

```
// Gera a URL /contact_manager/css/styles.css
echo $this->Html->css('ContactManager.styles');

// Gera a URL /contact_manager/js/widget.js
echo $this->Html->script('ContactManager.widget');

// Gera a URL /contact_manager/img/logo.jpg
echo $this->Html->image('ContactManager.logo');
```

Os recursos do plugin são servidos usando o filtro `AssetFilter` dispatcher por padrão. Isso só é recomendado para o desenvolvimento. Na produção, você deve *symlink do plugin symlink* para melhorar o desempenho.

Se você não estiver usando os helpers, você pode `/plugin_name/` para o início da URL para um recurso dentro desse plugin para atendê-lo. Ligando para `‘/contact_manager/js/some_file.js’` serviria o asset **plugins/ContactManager/webroot/js/some_file.js**.

Components, Helpers and Behaviors

Um plugin pode ter Components, Helpers e Behaviors, como uma aplicação CakePHP normal. Você pode até criar plugins que consistem apenas em Componentes, Helpers ou Behaviors que podem ser uma ótima maneira de construir componentes reutilizáveis que pode ser lançado em qualquer projeto.

Construir esses componentes é exatamente o mesmo que construí-lo dentro de uma aplicação normal, sem convenção de nome especial.

Referir-se ao seu componente de dentro ou fora do seu plugin requer apenas que você prefixa o nome do plugin antes do nome do componente. Por exemplo:

```
// Component definido no 'ContactManager' plugin
namespace ContactManager\Controller\Component;

use Cake\Controller\Component;

class ExampleComponent extends Component
{
}

// Dentro de seus controllers
public function initialize()
{
    parent::initialize();
    $this->loadComponent('ContactManager.Example');
}
```

A mesma técnica se aplica aos Helpers e Behaviors.

Expanda seu plugin

Este exemplo criou um bom começo para um plugin, mas há muito mais que você pode fazer. Como regra geral, qualquer coisa que você possa fazer com o seu aplicativo que você pode fazer dentro de um plugin também.

Vá em frente - inclua algumas bibliotecas de terceiros em ‘vendor’, adicione algumas novas shells para o cake console e não se esqueça de criar os testes então seus usuários de plugins podem testar automaticamente a funcionalidade do seu plugin!

Em nosso exemplo do ContactManager, podemos criar as actions add/remove/edit/delete no ContactsController, implementar a validação no model e implementar a funcionalidade que se poderia esperar ao gerenciar seus contatos. Depende de você decidir o que implementar no seu Plugins. Apenas não esqueça de compartilhar seu código com a comunidade, então que todos possam se beneficiar de seus componentes incríveis e reutilizáveis!

Publique seu plugin

Certifique-se de adicionar o seu plug-in para [Plugins.cakephp.org](https://plugins.cakephp.org)¹³⁸. Desta forma, outras pessoas podem Use-o como dependência do compositor. Você também pode propor seu plugin para o Lista de [awesome-cakephp list](#)¹³⁹.

Escolha um nome semanticamente significativo para o nome do pacote. Isso deve ser ideal prefixado com a dependência, neste caso “cakephp” como o framework. O nome do vendor geralmente será seu nome de usuário do GitHub. Não **não** use o espaço de nome CakePHP (cakephp), pois este é reservado ao CakePHP Plugins de propriedade.

A convenção é usar letras minúsculas e traços como separador.

Então, se você criou um plugin “Logging” com sua conta do GitHub “FooBar”, um bom nome seria *foo-bar/cakephp-logging*. E o plugin “Localized” do CakePHP pode ser encontrado em *cakephp/localized* respectivamente.

¹³⁸ <https://plugins.cakephp.org>

¹³⁹ <https://github.com/FriendsOfCake/awesome-cakephp>

REST

Muitos programadores de aplicações mais recentes estão percebendo a necessidade de abrir seu núcleo de funcionalidade para uma maior audiência. Fornecer acesso fácil e irrestrito ao seu a API principal pode ajudar a aceitar sua plataforma, e permite o mashups e fácil integração com outros sistemas.

Embora existam outras soluções, o REST é uma ótima maneira de facilitar o acesso a lógica que você criou em sua aplicação. É simples, geralmente baseado em XML (nós estamos falando XML simples, nada como um envelope SOAP) e depende de cabeçalhos HTTP para direção. Expor uma API via REST no CakePHP é simples.

A Configuração é simples

A maneira mais rápida de começar com o REST é adicionar algumas linhas para configurar *resource routes* em seu `config/routes.php`.

Uma vez que o roteador foi configurado para mapear solicitações REST para determinado controller as actions, podemos avançar para criar a lógica em nossas actions no controller. Um controller básico pode parecer algo assim:

```
// src/Controller/RecipesController.php
class RecipesController extends AppController
{

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function index()
    {
        $recipes = $this->Recipes->find('all');
        $this->set([
            'recipes' => $recipes,
            '_serialize' => ['recipes']
        ]);
    }

    public function view($id)
    {

```

```
$recipe = $this->Recipes->get($id);
$this->set([
    'recipe' => $recipe,
    '_serialize' => ['recipe']
]);
}

public function add()
{
    $recipe = $this->Recipes->newEntity($this->request->getData());
    if ($this->Recipes->save($recipe)) {
        $message = 'Saved';
    } else {
        $message = 'Error';
    }
    $this->set([
        'message' => $message,
        'recipe' => $recipe,
        '_serialize' => ['message', 'recipe']
    ]);
}

public function edit($id)
{
    $recipe = $this->Recipes->get($id);
    if ($this->request->is(['post', 'put'])) {
        $recipe = $this->Recipes->patchEntity($recipe, $this->request->getData());
        if ($this->Recipes->save($recipe)) {
            $message = 'Saved';
        } else {
            $message = 'Error';
        }
    }
    $this->set([
        'message' => $message,
        '_serialize' => ['message']
    ]);
}

public function delete($id)
{
    $recipe = $this->Recipes->get($id);
    $message = 'Deleted';
    if (!$this->Recipes->delete($recipe)) {
        $message = 'Error';
    }
    $this->set([
        'message' => $message,
        '_serialize' => ['message']
    ]);
}
}
```

Os controllers RESTful geralmente usam extensões analisadas para exibir diferentes visualizações com base em diferentes tipos de requisições. Como estamos lidando com pedidos REST, estaremos fazendo visualizações XML. Você pode fazer visualizações JSON usando o CakePHP's para criar *Views JSON & XML*. Ao usar o build-in `XmlView` podemos definir uma variável na view `_serialize`. A variável de exibição é usada para definir quais variáveis de exibição `XmlView` devem Serializar em XML ou JSON.

Se quisermos modificar os dados antes de serem convertidos em XML ou JSON, não devemos definir a variável de exibição “`_serialize`” e, em vez disso, use arquivos de template. Colocamos as saídas REST para nosso `RecipesController` dentro de `src/Template/Recipes/xml`. Nós também podemos usar `TheXml` para saída XML rápida e fácil:

```
// src/Template/Recipes/xml/index.ctp
// Faça alguma formatação e manipulação em
// the $recipes array.
$xml = Xml::fromArray(['response' => $recipes]);
echo $xml->asXML();
```

Ao servir um tipo de conteúdo específico usando `Cake\Routing\Router::extensions()`, CakePHP procura automaticamente um auxiliar de visualização. Uma vez que estamos usando o XML como o tipo de conteúdo, não há um helper interno, no entanto, se você criasse um, ele seria automaticamente carregado Para o nosso uso nessas views.

O XML renderizado acabará por parecer algo assim:

```
<recipes>
  <recipe>
    <id>234</id>
    <created>2008-06-13</created>
    <modified>2008-06-14</modified>
    <author>
      <id>23423</id>
      <first_name>Billy</first_name>
      <last_name>Bob</last_name>
    </author>
    <comment>
      <id>245</id>
      <body>Yummy yummy</body>
    </comment>
  </recipe>
  ...
</recipes>
```

Criar uma lógica para editar uma action é um pouco mais complicado, mas não muito. Desde a que você está fornecendo uma API que emite XML, é uma escolha natural para receber XML Como entrada. Não se preocupe, o `Cake\Controller\Component\RequestHandler` e `Cake\Routing\Router` tornam as coisas muito mais fáceis. Se um POST ou A solicitação PUT tem um tipo de conteúdo XML, então a entrada é executada através do CakePHP's `Xml`, e a representação da array dos dados é atribuída a `$this->request->getData()`. Devido a essa característica, lidar com dados XML e POST em O paralelo é transparente: não são necessárias alterações ao código do controlador ou do modelo. Tudo o que você precisa deve terminar em `$this->request->getData()`.

Aceitando entrada em outros formatos

Normalmente, os aplicativos REST não apenas exibem conteúdo em formatos de dados alternativos, Mas também aceitam dados em diferentes formatos. No CakePHP, o `RequestHandlerComponent` ajuda a facilitar isso. Por padrão, Ele decodificará qualquer entrada de dados de entrada JSON/XML para solicitações POST/PUT E forneça a versão da array desses dados em `$this->request->getData()`. Você também pode usar desserializadores adicionais para formatos alternativos se você Precisa deles, usando `RequestHandler::addInputType()`.

Roteamento RESTful

O roteador CakePHP facilita a conexão das rotas de recursos RESTful. Veja a seção *Criando rotas RESTful* para mais informações.

Segurança

O CakePHP fornece algumas ferramentas para proteger sua aplicação. As seguintes seções abrangem essas ferramentas:

Segurança

`class Cake\Utility\Security`

A [biblioteca de segurança](#)¹⁴⁰ trabalha com medidas básicas de segurança fornecendo métodos para *hash* e criptografia de dados.

Criptografando e Descriptografando Dados

`static Cake\Utility\Security::encrypt($text, $key, $hmacSalt = null)`

`static Cake\Utility\Security::decrypt($cipher, $key, $hmacSalt = null)`

Criptografando `$text` usando AES-256. A `$key` deve ser um valor com dados variados como uma senha forte. O resultado retornado será o valor criptografado com um *HMAC checksum*.

Este método irá usar `openssl`¹⁴¹ ou `mcrypt`¹⁴² dependendo de qual deles estiver disponível em seu sistema. Dados criptografados em uma implementação são portáveis para a outra.

Aviso: A extensão `mcrypt`¹⁴³ foi considerada obsoleta no PHP7.1

¹⁴³ <http://php.net/mcrypt>

Este método **nunca** deve ser usado para armazenar senhas. Em vez disso, você deve usar o método de *hash* de mão única fornecidos por `Utility\Security::hash()`. Um exemplo de uso pode ser:

```
// Assumindo que $key está gravada em algum lugar, ela pode ser reusada para
// descriptografar depois.
```

¹⁴⁰ <https://api.cakephp.org/3.x/class-Cake.Utility.Security.html>

¹⁴¹ <http://php.net/openssl>

¹⁴² <http://php.net/mcrypt>

```
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';
$result = Security::encrypt($value, $key);
```

Se você não fornecer um HMAC salt, o valor em `Security.salt` será usado. Os valores criptografados podem ser descriptografados usando `Cake\Utility\Security::decrypt()`.

Descriptografando um valor criptografado anteriormente. Os parametros `$key` e `$hmacSalt` devem corresponder aos valores utilizados para a criptografia senão o processo falhará.

Exemplo:

```
// Assumindo que $key está gravada em algum lugar, ela pode ser reusada para
// descriptografar depois.
$key = 'wt1U5MACWJFTXGenFoZoiLwQGrLgdbHA';

$cipher = $user->secrets;
$result = Security::decrypt($cipher, $key);
```

Se o valor não puder ser descriptografado por mudanças em `$key` ou HMAC salt o método retornará `false`.

Escolhendo uma implementação de criptografia

Se você está atualizando sua aplicação do CakePHP 2.x, os dados criptografados não serão compatíveis com openssl por que seus dados criptografados não são totalmente compatíveis com AES. Se você não quiser ter o trabalho de refazer a criptografia dos seus dados, você pode forçar o CakePHP a usar `mcrypt` através do método `engine()`:

```
// Em config/bootstrap.php
use Cake\Utility\Crypto\Mcrypt;

Security::engine(new Mcrypt());
```

O código acima permitirá que você leia dados de versões anteriores do CakePHP e criptografar novos dados para serem compatíveis com OpenSSL.

Fazendo Hash de dados

static `Cake\Utility\Security::hash($string, $type = NULL, $salt = false)`

Criando um hash de uma string usando o método acima. Se `$salt` estiver setado como `true` o salt da aplicação será utilizado:

```
// Usando salt definido na aplicação
$shal = Security::hash('CakePHP Framework', 'sha1', true);

// Usando um salt customizado
$shal = Security::hash('CakePHP Framework', 'sha1', 'my-salt');

// Usando o padrão do algoritmo de hash
$hash = Security::hash('CakePHP Framework');
```

O método `hash()` suporta as seguintes estratégias de hash:

- md5
- sha1
- sha256

E qualquer outro algoritmo de hash que a função `hash()` do PHP suporta.

Aviso: Você não deve usar `hash()` para senhas em novas aplicações, o ideal é usar a classe `DefaultPasswordHasher` que usa `bcrypt` por padrão.

Gerando dados aleatórios seguros

static `Cake\Utility\Security::randomBytes($length)`

Obter `$length` número de bytes de uma fonte segura aleatória. Esta função utiliza um dos seguintes métodos:

- Função `random_bytes` do PHP.
- Função `openssl_random_pseudo_bytes` da extensão SSL.

Se nenhuma das opções estiverem disponíveis um `warning` será emitido e um valor não seguro será usado por motivos de compatibilidade.

Novo na versão 3.2.3: O método `randomBytes` foi adicionado na versão 3.2.3.

Sessions

Session Configuration

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁴⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁴⁴ <https://github.com/cakephp/docs>

Testing

O CakePHP vem com suporte interno para testes e integração para o [PHPUnit](#)¹⁴⁵. Em adição aos recursos oferecidos pelo PHPUnit, o CakePHP oferece alguns recursos adicionais para fazer testes mais facilmente. Esta seção abordará a instalação do PHPUnit, começando com testes unitários e como você pode usar as extensões que o CakePHP oferece.

Instalando o PHPUnit

O CakePHP usa o PHPUnit como framework de teste básico. O PHPUnit é um padrão para testes unitários em PHP. Ele oferece um profundo e poderoso conjunto de recursos para você ter certeza que o seu código faz o que você acha que ele faz. O PHPUnit pode ser instalado usando o [PHAR package](#)¹⁴⁶ ou [Composer](#)¹⁴⁷.

Instalando o PHPUnit com Composer

Para instalar o PHPUnit com Composer:

```
$ php composer.phar require --dev phpunit/phpunit
```

Isto adicionará a dependência para a seção `require-dev` do seu `composer.json`, e depois instalará o PHPUnit com qualquer outra dependência.

Agora você executa o PHPUnit usando:

```
$ vendor/bin/phpunit
```

Usando o arquivo PHAR

Depois de ter baixado o arquivo **phpunit.phar**, você pode usar ele para executar seus testes:

```
php phpunit.phar
```

¹⁴⁵ <http://phpunit.de>

¹⁴⁶ <http://phpunit.de/#download>

¹⁴⁷ <http://getcomposer.org>

Dica: Como conveniência você pode deixar `phpunit.phar` disponível globalmente em sistemas Unix ou Linux com os comandos:

```
chmod +x phpunit.phar
sudo mv phpunit.phar /usr/local/bin/phpunit
phpunit --version
```

Por favor, consulte a documentação do PHPUnit para instruções sobre como instalar globalmente o PHPUnit PHAR em sistemas Windows¹⁴⁸.

Configuração do banco de dados test

Lembre-se de ter o debug abilitado em seu arquivo **config/app.php** antes de executar qualquer teste. Antes de executar quaisquer testes você deve adicionar um datasource `test` para o arquivo **config/app.php**. Esta configuração é usada pelo CakePHP para fixar tabelas e dados:

```
'Datasources' => [
    'test' => [
        'datasource' => 'Cake\Database\Driver\Mysql',
        'persistent' => false,
        'host' => 'dbhost',
        'username' => 'dblogin',
        'password' => 'dbpassword',
        'database' => 'test_database'
    ],
],
```

Running Tests

Fixtures

Controller Integration Testing

Testing Actions That Require Authentication

Testing Events

Nota: É uma boa ideia usar bancos de dados diferentes para o banco de testes e para o banco de desenvolvimento. Isto evitara erros mais tarde.

¹⁴⁸ <http://phpunit.de/manual/current/en/installation.html#installation.phar.windows>

Validação

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁴⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁴⁹ <https://github.com/cakephp/docs>

App Class

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁰ <https://github.com/cakephp/docs>

Collections (Coleções)

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵¹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵¹ <https://github.com/cakephp/docs>

Arquivos & Pastas

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵² ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵² <https://github.com/cakephp/docs>

Hash

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵³ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵³ <https://github.com/cakephp/docs>

Http Client

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁴ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁴ <https://github.com/cakephp/docs>

Inflector

class Cake\Utility\Inflector

A classe `Inflector` recebe uma string e a manipula afim de suportar variações de palavras como pluralizações ou CamelCase e normalmente é acessada estaticamente. Exemplo: `Inflector::pluralize('example')` retorna “examples”.

Você pode testar as inflexões em inflector.cakephp.org¹⁵⁵.

Resumo dos métodos de Inflexão e Suas Saídas

Resumo rápido dos métodos embutidos no Inflector e os resultados que produzem quando fornecidos um argumento de palavra composta.

Method	Argument	Output
pluralize()	BigApple	BigApples
	big_apple	big_apples
singularize()	BigApples	BigApple
	big_apples	big_apple
camelize()	big_apples	BigApples
	big apple	BigApple
underscore()	BigApples	big_apples
	Big Apples	big apples
humanize()	big_apples	Big Apples
	bigApple	BigApple
classify()	big_apples	BigApple
	big apple	BigApple
dasherize()	BigApples	big-apples
	big apple	big apple
tableize()	BigApple	big_apples
	Big Apple	big apples
variable()	big_apple	bigApple
	big apples	bigApples
slug()	Big Apple	big-apple
	BigApples	BigApples

¹⁵⁵ <https://inflector.cakephp.org/>

Criando as formas singulares e plurais

```
static Cake\Utility\Inflector::singularize($singular)
```

```
static Cake\Utility\Inflector::pluralize($singular)
```

Tanto `pluralize()` quanto `singularize()` funcionam para a maioria dos substantivos do Inglês. Caso seja necessário o suporte para outras línguas, você pode usar *Configuração da inflexão* para personalizar as regras usadas:

```
// Apples
echo Inflector::pluralize('Apple');
```

Nota: `pluralize()` pode não funcionar corretamente nos casos onde um substantivo já esteja em sua forma plural.

```
// Person
echo Inflector::singularize('People');
```

Nota: `singularize()` pode não funcionar corretamente nos casos onde um substantivo já esteja em sua forma singular.

Criando as formas CamelCase e nome_sublinhado

```
static Cake\Utility\Inflector::camelize($underscored)
```

```
static Cake\Utility\Inflector::underscore($camelCase)
```

Estes métodos são úteis para a criação de nomes de classe ou de propriedades:

```
// ApplePie
Inflector::camelize('Apple_pie')

// apple_pie
Inflector::underscore('ApplePie');
```

É importante ressaltar que `underscore()` irá converter apenas palavras formatadas em CamelCase. Palavras com espaços serão convertidas para caixa baixa, mas não serão separadas por sublinhado.

Criando formas legíveis para humanos

```
static Cake\Utility\Inflector::humanize($underscored)
```

Este método é útil para converter da forma sublinhada para o “Formato Título” para a leitura humana:

```
// Apple Pie
Inflector::humanize('apple_pie');
```

Criando formatos para nomes de tabelas e classes

```
static Cake\Utility\Inflector::classify($underscored)
```

```
static Cake\Utility\Inflector::dasherize($dashed)
```

```
static Cake\Utility\Inflector::tableize($camelCase)
```

Ao gerar o código ou usar as convenções do CakePHP, você pode precisar inferir os nomes das tabelas ou classes:

```
// UserProfileSettings
Inflector::classify('user_profile_settings');

// user-profile-setting
Inflector::dasherize('UserProfileSetting');

// user_profile_settings
Inflector::tableize('UserProfileSetting');
```

Criando nomes de variáveis

```
static Cake\Utility\Inflector::variable($underscored)
```

Nomes de variáveis geralmente são úteis em tarefas de meta-programação que envolvem a geração de código ou rotinas baseadas em convenções:

```
// applePie
Inflector::variable('apple_pie');
```

Criando strings de URL seguras

```
static Cake\Utility\Inflector::slug($word, $replacement = '-')
```

`slug()` converte caracteres especiais em suas versões normais e converte os caracteres não encontrados e espaços em traços. O método `slug()` espera que a codificação seja UTF-8:

```
// apple-puree
Inflector::slug('apple purée');
```

Nota: `Inflector::slug()` foi depreciado desde a versão 3.2.7. Procure usar `Text::slug()` de agora em diante.

Configuração da inflexão

As convenções de nomes do CakePHP podem ser bem confortáveis. Você pode nomear sua tabela no banco de dados como `big_boxes`, seu modelo como `BigBoxes`, seu controlador como `BigBoxesController` e tudo funcionará automaticamente. O CakePHP entrelaça todos estes conceitos através da inflexão das palavras em suas formas singulares e plurais.

Porém ocasionalmente (especialmente para os nossos amigos não Anglófonos) podem encontrar situações onde o infletor do CakePHP (a classe que pluraliza, singulariza, transforma em CamelCase e em nome_sublinhado) não funciona como você gostaria. Caso o CakePHP não reconheça seu “quaisquer” ou “lápiz”, você pode ensiná-lo a entender seus casos especiais.

Carregando inflexões personalizadas

static `Cake\Utility\Inflector::rules($type, $rules, $reset = false)`

Define novas inflexões e transliteraões para o `Inflector` usar. Geralmente este método deve ser chamado no seu **config/bootstrap.php**:

```
Inflector::rules('singular', ['/^(bil)er$/i' => '\1', '/^(inflec|contribu)tors$/i' =>
↳ '\1ta']);
Inflector::rules('uninflected', ['singulars']);
Inflector::rules('irregular', ['phylum' => 'phyla']); // The key is singular form,
↳ value is plural form
```

As regras ditadas por este método serão agregadas aos conjuntos de inflexão definidos em `Cake/Utility/Inflector`, onde elas terão prioridade sobre as regras já declaradas por padrão. Você pode usar `Inflector::reset()` para limpar todas as regras e retornar o `Inflector` para seu estado original.

Número

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁶ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁶ <https://github.com/cakephp/docs>

Objetos de Registro

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁷ <https://github.com/cakephp/docs>

Texto

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁸ <https://github.com/cakephp/docs>

Tempo

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁵⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁵⁹ <https://github.com/cakephp/docs>

Xml

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁶⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁶⁰ <https://github.com/cakephp/docs>

Constantes e Funções

A maior parte do seu trabalho diário com o CakePHP será feito utilizando classes e métodos do *core*. O CakePHP disponibiliza funções globais de conveniência que podem ajudar. Muitas dessas funções são usadas em classes do CakePHP (carregando um *model* ou um *component*), mas outras tornam mais fácil o trabalho de lidar com *arrays* ou *strings*.

Nós também vamos cobrir algumas das constantes existentes em aplicações CakePHP. Constantes essas, que facilitam *upgrades* e apontam convenientemente para arquivos e diretórios chaves da sua aplicação.

Funções globais

Aqui estão as funções disponíveis globalmente no CakePHP. A maioria delas são *wrappers* de conveniência para funcionalidades do CakePHP, como por exemplo, *debug* e localização de conteúdo.

___ (string \$string_id[, \$formatArgs])

Essa função lida com a localização da sua aplicação. O `$string_id` identifica o ID usado para a tradução. *Strings* são tratadas seguindo o formato usado no `sprintf()`. Você pode fornecer argumentos adicionais para substituir *placeholders* na sua string:

```
___('Você tem {0} mensagens', $number);
```

Nota: Verifique a seção [Internacionalização e Localização](#) para mais informações.

___d (string \$domain, string \$msg, mixed \$args = null)

Permite sobrescrever o domínio atual por uma mensagem simples.

Muito útil ao localizar um *plugin*: `echo ___d('PluginName', 'Esse é meu plugin');`

___dn (string \$domain, string \$singular, string \$plural, integer \$count, mixed \$args = null)

Permite sobrescrever o domínio atual por uma mensagem no plural. Retorna a forma correta da mensagem no plural identificada por `$singular` e `$plural`, pelo contador `$count` e pelo domínio `$domain`.

___dx (string \$domain, string \$context, string \$msg, mixed \$args = null)

Permite sobrescrever o domínio atual por uma mensagem simples. Também permite a especificação de um contexto.

O contexto é um identificador único para as *strings* de tradução que a tornam únicas sob um mesmo domínio.

—**dxn** (*string \$domain, string \$context, string \$singular, string \$plural, integer \$count, mixed \$args = null*)

Permite sobrescrever o domínio atual por uma mensagem no plural. Também permite a especificação de um contexto. Retorna a forma correta da mensagem no plural identificada por *\$singular* e *\$plural*, pelo contador *\$count* e pelo domínio *\$domain*. Alguns idiomas tem mais de uma forma para o plural dependendo do contador.

O contexto é um identificador único para as *strings* de tradução que a tornam únicas sob um mesmo domínio.

—**n** (*string \$singular, string \$plural, integer \$count, mixed \$args = null*)

Retorna a forma correta da mensagem no plural identificada por *\$singular* e *\$plural*, pelo contador *\$count* e pelo domínio *\$domain*. Alguns idiomas tem mais de uma forma para o plural dependendo do contador.

—**x** (*string \$context, string \$msg, mixed \$args = null*)

O contexto é um identificador único para as *strings* de tradução que a tornam únicas sob um mesmo domínio.

—**xn** (*string \$context, string \$singular, string \$plural, integer \$count, mixed \$args = null*)

Retorna a forma correta da mensagem no plural identificada por *\$singular* e *\$plural*, pelo contador *\$count* e pelo domínio *\$domain*. Alguns idiomas tem mais de uma forma para o plural dependendo do contador.

O contexto é um identificador único para as *strings* de tradução que a tornam únicas sob um mesmo domínio.

collection (*mixed \$items*)

Wrapper de conveniência para instanciar um novo objeto `Cake\Collection\Collection`, re-passando o devido argumento. O parâmetro *\$items* recebe tanto um objeto `Traversable` quanto um *array*.

debug (*mixed \$var, boolean \$showHtml = null, \$showFrom = true*)

Alterado na versão 3.3.0: Esse método retorna a *\$var* passada para que você possa, por instância, colocá-la em uma declaração de retorno.

Se a variável do core `$debug` for `true`, *\$var* será imprimida. Se `$showHTML` for `true`, ou for deixada como `null` os dados serão renderizados formatados para melhor exibição em navegadores. Se `$showFrom` não for definida como `false`, o *debug* começará a partir da linha em que foi chamado. Também veja [Depuração](#)

pr (*mixed \$var*)

Alterado na versão 3.3.0: Chamar esse método vai retornar a *\$var* passada, então, você pode, por instância, colocá-la em uma declaração de retorno.

Wrapper de conveniência para `print_r()` com a adição das *tags* `<pre>` ao redor da saída.

pj (*mixed \$var*)

Alterado na versão 3.3.0: Chamar esse método vai retornar a *\$var* passada, então, você pode, por instância, colocá-la em uma declaração de retorno.

Função de conveniência para formatação de JSON, com a adição das *tags* `<pre>` ao redor da saída.

Deve ser usada com o intuito de *debugar* JSON de objetos e *arrays*.

env (*string \$key, string \$default = null*)

Alterado na versão 3.1.1: O parâmetro *\$default* será adicionado.

Recebe uma variável de ambiente de fontes disponíveis. Usada como *backup* se `$_SERVER` ou `$_ENV` estiverem desabilitados.

Essa função também emula `PHP_SELF` e `DOCUMENT_ROOT` em servidores não suportados. De fato, é sempre uma boa ideia usar `env()` ao invés de `$_SERVER` ou `getenv()` (especialmente se você planeja distribuir o código), pois é um *wrapper* completo de emulação.

h (*string \$text, boolean \$double = true, string \$charset = null*)

Wrapper de conveniência para `htmlspecialchars()`.

pluginSplit (*string \$name, boolean \$dotAppend = false, string \$plugin = null*)

Divide um nome de plugin que segue o padrão de sintaxe de pontos e o transforma em um nome de classe ou do *plugin*. Se *\$name* não tem um ponto, então o índice 0 será *null*.

Comumente usada assim: `list($plugin, $name) = pluginSplit('Users.User');`

namespaceSplit (*string \$class*)

Divide o *namespace* do nome da classe.

Comumente usada assim: `list($namespace, $className) = namespaceSplit('Cake\Core\App');`

Constantes de definição do Core

A maior parte das constantes a seguir referem-se a caminhos da sua aplicação.

constant APP

Caminho absoluto para o diretório de sua aplicação, incluindo a barra final.

constant APP_DIR

Igual a `app` ou ao nome do diretório de sua aplicação.

constant CACHE

Caminho para o diretório de arquivos de cache. Pode ser compartilhado entre hosts em uma configuração multi-servidores.

constant CAKE

Caminho para o diretório do CakePHP.

constant CAKE_CORE_INCLUDE_PATH

Caminho para o diretório raiz de bibliotecas.

constant CONFIG

Caminho para o diretório de configurações.

constant CORE_PATH

Caminho para o diretório raiz com contra-barras no final.

constant DS

Atalho para o `DIRECTORY_SEPARATOR` do PHP, que é `/` no Linux e `\\` no Windows.

constant LOGS

Caminho para o diretório de logs.

constant ROOT

Caminho para o diretório raiz.

constant TESTS

Caminho para o diretório de testes.

constant TMP

Caminho para o diretório de arquivos temporários.

constant WWW_ROOT

Caminho completo para o diretório webroot.

Constantes de definição de tempo

constant TIME_START

Timestamp unix em microsegundos como *float* de quando a aplicação começou.

constant SECOND

Igual a 1

constant MINUTE

Igual a 60

constant HOUR

Igual a 3600

constant DAY

Igual a 86400

constant WEEK

Igual a 604800

constant MONTH

Igual a 2592000

constant YEAR

Igual a 31536000

Debug Kit

DebugKit é um plugin suportado pelo time principal do CakePHP que oferece uma barra de ferramentas para auxiliar na depuração de aplicações do CakePHP.

Instalação

Por padrão o DebugKit é instalado com o esqueleto padrão da aplicação. Se você o removeu e gostaria de reinstalá-lo, você pode executar o seguinte comando a partir do diretório raiz da aplicação (onde o arquivo `composer.json` está localizado):

```
php composer.phar require --dev cakephp/debug_kit "~3.0"
```

Então, você precisará habilitar o plugin ao executar o seguinte comando:

```
bin/cake plugin load DebugKit
```

Armazenamento do DebugKit

Por padrão, o DebugKit usa um pequeno banco de dados SQLite no diretório `/tmp` de sua aplicação para armazenar os dados referentes ao painel de informações. Se você quiser que o DebugKit armazene seus dados em outro lugar, é necessário configurar uma nova conexão com o nome `debug_kit`.

Configuração do banco de dados

Como informado anteriormente, por padrão, o DebugKit armazenará os dados do painel em um banco de dados SQLite no diretório `/tmp` de sua aplicação. Se você não puder instalar a extensão `pdo_sqlite` do PHP, você pode configurar o DebugKit para usar um banco de dados diferente ao definir uma conexão `debug_kit` em seu arquivo **`config/app.php`**. Por exemplo:

```
/**
 * A conexão debug_kit armazena meta-dados do DebugKit.
 */
'debug_kit' => [
```

```
'className' => 'Cake\Database\Connection',
'driver' => 'Cake\Database\Driver\Mysql',
'persistent' => false,
'host' => 'localhost',
// 'port' => 'nonstandard_port_number',
'username' => 'dbusername',    // Your DB username here
'password' => 'dbpassword',    // Your DB password here
'database' => 'debug_kit',
'encoding' => 'utf8',
'timezone' => 'UTC',
'cacheMetadata' => true,
'quoteIdentifiers' => false,
// 'init' => ['SET GLOBAL innodb_stats_on_metadata = 0'],
],
```

Uso da barra de ferramentas

A barra de ferramentas do DebugKit é composta por vários painéis, que são exibidos ao clicar no ícone do CakePHP no canto inferior direito da janela do seu navegador. Uma vez que a barra de ferramentas é aberta, você deverá ver uma série de botões. Cada um desses botões se expande em um painel de informações relacionadas.

Cada painel permite a você observar aspectos diferentes da sua aplicação:

- **Cache** Exibe o uso de cache durante uma solicitação e limpa caches.
- **Environment** Exibe variáveis de ambiente relacionadas com PHP + CakePHP.
- **History** Exibe uma lista de requisições anteriores, e permite que você carregue e veja dados da barra de ferramentas a partir de solicitações anteriores.
- **Include** Exibe os arquivos incluídos divididos por grupo.
- **Log** Exibe todas as entradas feitas nos arquivos de log deste pedido.
- **Request** Exibe informações sobre a requisição atual, GET, POST, informações sobre a rota atual do Cake e Cookies.
- **Session** Exibe a informação atual da sessão.
- **Sql Logs** Exibe logs SQL para cada conexão com o banco de dados.
- **Timer** Exibe qualquer temporizador que fora definido durante a requisição com `DebugKit\DebugTimer`, e memória utilizada coletada com `DebugKit\DebugMemory`.
- **Variables** Exibe variáveis de View definidas em um Controller.

Tipicamente, um painel manipula a recolha e apresentação de um único tipo de informações como logs ou informações de requisições. Você pode optar por visualizar os painéis padrões da barra de ferramentas ou adicionar seus próprios painéis personalizados.

Usando o painel History

O painel History é uma das características mais frequentemente confundidas do DebugKit. Ele oferece uma forma de visualizar os dados de requisições anteriores na barra de ferramentas, incluindo erros e redirecionamentos.


Como você pode ver, o painel contém uma lista de requisições. Na esquerda você pode ver um ponto marcando a requisição ativa. Clicar em qualquer requisição vai carregar os dados referentes a mesma no painel. Quando os

History ×

10 previous requests available

- Back to current request

1/11/15, 2:02 AM	GET 404 text/html /bookmarks/derps
1/11/15, 2:01 AM	GET 200 text/html /bookmarks
1/11/15, 2:01 AM	GET 200 text/html /users
1/11/15, 2:00 AM	GET 200 text/html /bookmarks/
1/11/15, 2:00 AM	GET 200 text/html /bookmarks/
1/11/15, 2:00 AM	GET 200 text/html /bookmarks/
1/11/15, 2:00 AM	GET 200 text/html /bookmarks/

Cache Environment **History** Include Log 6 Request Session Sql Log 6 - 3 ms 

dados são carregados, os títulos do painel vão sofrer uma transição para indicar que informações alternativas foram carregados.

Desenvolvendo seus próprios painéis

Você pode criar seus próprios painéis customizados do DebugKit para ajudar na depuração de suas aplicações.

Criando uma Panel Class

Panel Classes precisam ser colocadas no diretório **src/Panel**. O nome do arquivo deve combinar com o nome da classe, então a classe `MyCustomPanel` deveria remeter ao nome de arquivo **src/Panel/MyCustomPanel.php**:

```
namespace App\Panel;

use DebugKit\DebugPanel;

/**
 * My Custom Panel
 */
class MyCustomPanel extends DebugPanel
{
    ...
}
```

Perceba que painéis customizados são necessários para estender a classe `DebugPanel`.

Callbacks

Por padrão objetos do painel possuem dois callbacks, permitindo-lhes acoplar-se na requisição atual. Painéis inscrevem-se aos eventos `Controller.initialize` e `Controller.shutdown`. Se o seu painel precisa inscrever-se a eventos adicionais, você pode usar o método `implementedEvents` para definir todos os eventos aos quais o seu painel possa precisar estar inscrito.

Você deveria estudar os painéis nativos para absorver alguns exemplos de como construir painéis.

Elementos do painel

Cada painel deve ter um elemento view que renderiza o conteúdo do mesmo. O nome do elemento deve ser sublinhado e flexionado a partir do nome da classe. Por exemplo `SessionPanel` deve possuir um elemento nomeado `session_panel.ctp`, e `SqllogPanel` deve possuir um elemento nomeado `sqllog_panel.ctp`. Estes elementos devem estar localizados na raiz do seu diretório `src/Template/Element`.

Títulos personalizados e Elementos

Os painéis devem relacionar o seu título e o nome do elemento por convenção. No entanto, se você precisa escolher um nome de elemento personalizado ou título, você pode definir métodos para customizar o comportamento do seu painel:

- `title()` - Configure o título que é exibido na barra de ferramentas.
- `elementName()` - Configure qual elemento deve ser utilizada para um determinado painel.

Métodos de captura

Você também pode implementar os seguintes métodos para customizar como o seu painel se comporta e se aparenta:

- `shutdown(Event $event)` Esse método coleta e prepara os dados para o painel. Os dados são geralmente armazenados em `$this->_data`.
- `summary()` Este método retorna uma *string* de dados resumidos para serem exibidos na *toolbar*, mesmo quando um painel está minimizado. Frequentemente, é um contador ou um pequeno resumo de informações.
- `data()` Este método retorna os dados do painel que serão usados como contexto para o elemento. Você pode manipular os dados coletados no método `shutdown()`. Esse método **deve** retornar dados que podem ser serializados.

Painéis em outros plugins

Painéis disponibilizados por *Plugins* funcionam quase que totalmente como outros plugins, com uma pequena diferença: Você deve definir `public $plugin` como o nome do diretório do plugin, com isso os elementos do painel poderão ser encontrados no momento de renderização:

```
namespace MyPlugin\Panel;

use DebugKit\DebugPanel;

class MyCustomPanel extends DebugPanel
{
    public $plugin = 'MyPlugin';
}
```

```
    ...  
}
```

Para usar um plugin ou painel da aplicação, atualize a configuração do DebugKit de sua aplicação para incluir o painel:

```
Configure::write(  
    'DebugKit.panels',  
    array_merge(Configure::read('DebugKit.panels'), ['MyCustomPanel'])  
);
```

O código acima deve carregar todos os painéis padrão tanto como os outros painéis customizados do MyPlugin.

Migrations

Migrations é um plugin suportado pela equipe oficial do CakePHP que ajuda você a fazer mudanças no **schema** do banco de dados utilizando arquivos PHP, que podem ser versionados utilizando um sistema de controle de versão.

Ele permite que você atualize suas tabelas ao longo do tempo. Ao invés de escrever modificações de **schema** via SQL, este plugin permite que você utilize um conjunto intuitivo de métodos para fazer mudanças no seu banco de dados.

Esse plugin é um **wrapper** para a biblioteca [Phinx](https://phinx.org/)¹⁶¹.

Instalação

Por padrão o plugin é instalado junto com o esqueleto da aplicação. Se você o removeu e quer reinstalá-lo, execute o comando a seguir a partir do diretório **ROOT** da sua aplicação (onde o arquivo `composer.json` está localizado):

```
$ php composer.phar require cakephp/migrations "@stable"

// Or if composer is installed globally

$ composer require cakephp/migrations "@stable"
```

Para usar o plugin você precisa carregá-lo no arquivo **config/bootstrap.php** da sua aplicação. Você pode usar o *shell de plugins do CakePHP* para carregar e descarregar plugins do seu arquivo **config/bootstrap.php**:

```
$ bin/cake plugin load Migrations
```

Ou você pode carregar o plugin editando seu arquivo **config/bootstrap.php** e adicionando a linha:

```
Plugin::load('Migrations');
```

Adicionalmente, você precisará configurar o banco de dados padrão da sua aplicação, no arquivo **config/app.php** como explicado na seção *Configuração de banco de dados*.

¹⁶¹ <https://phinx.org/>

Visão Geral

Uma migração é basicamente um arquivo PHP que descreve as mudanças a serem feitas no banco de dados. Um arquivo de migração pode criar ou excluir tabelas, adicionar ou remover colunas, criar índices e até mesmo inserir dados em seu banco de dados.

Aqui segue um exemplo de migração:

```
<?php
use Migrations\AbstractMigration;

class CreateProducts extends AbstractMigration
{
    /**
     * Change Method.
     *
     * More information on this method is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-change-method
     * @return void
     */
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('description', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

Essa migração irá adicionar uma tabela chamada `products` ao banco de dados com as seguintes colunas:

- `id` coluna do tipo `integer` como chave primária
- `name` coluna do tipo `string`
- `description` coluna do tipo `text`
- `created` coluna do tipo `datetime`
- `modified` coluna do tipo `datetime`

Dica: A coluna de chave primária `id` será adicionada **implicitamente**.

Nota: Note que este arquivo descreve como o banco de dados deve ser **após** a aplicação da migração. Neste ponto, a tabela `products` ainda não existe no banco de dados, nós apenas criamos um arquivo que é capaz de criar a tabela `products` com seus devidos campos ou excluir a tabela caso uma operação `rollback` seja executada.

Com o arquivo criado na pasta **config/Migrations**, você será capaz de executar o comando abaixo para executar as migrações no seu banco de dados:

```
bin/cake migrations migrate
```

O comando seguinte irá executar um **rollback** na migração e irá excluir a tabela recém criada:

```
bin/cake migrations rollback
```

Criando migrations

Arquivos de migração são armazenados no diretório **config/Migrations** da sua aplicação. O nome dos arquivos de migração têm como prefixo a data em que foram criados, no formato **YYYYMMDDHHMMSS_MigrationName.php**. Aqui estão exemplos de arquivos de migração:

- 20160121163850_CreateProducts.php
- 20160210133047_AddRatingToProducts.php

A maneira mais fácil de criar um arquivo de migrações é usando o *Geração de código com bake* a linha de comando.

Por favor, leia a *documentação do Phinx* <<http://docs.phinx.org/en/latest/migrations.html>> a fim de conhecer a lista completa dos métodos que você pode usar para escrever os arquivos de migração.

Nota: Ao gerar as migrações através do `bake` você ainda pode alterá-las antes da sua execução, caso seja necessário.

Sintaxe

A sintaxe do `bake` para a geração de migrações segue o formato abaixo:

```
$ bin/cake bake migration CreateProducts name:string description:text created modified
```

Quando utilizar o `bake` para criar as migrações, você normalmente precisará informar os seguintes dados:

```
* o nome da migração que você irá gerar (`CreateProducts` por exemplo)
* as colunas da tabela que serão adicionadas ou removidas na migração
(`name:string description:text created modified` no nosso caso)
```

Devido às convenções, nem todas as alterações de schema podem ser realizadas através destes comandos.

Além disso, você pode criar um arquivo de migração vazio caso deseje ter um controle total do que precisa ser executado. Para isto, apenas omita a definição das colunas:

```
$ bin/cake migrations create MyCustomMigration
```

Nomenclatura de migrations

A nomenclatura das migrações pode seguir qualquer um dos padrões apresentados a seguir:

- `(/^ (Create) (.*) /)` Cria a tabela especificada.
- `(/^ (Drop) (.*) /)` Exclui a tabela especificada. Ignora campos especificados nos argumentos
- `(/^ (Add) .* (? : To) (.*) /)` Adiciona campos a tabela especificada
- `(/^ (Remove) .* (? : From) (.*) /)` Remove campos de uma tabela específica
- `(/^ (Alter) (.*) /)` Altera a tabela especificada. Um apelido para um CreateTable seguido de um AlterTable

Você também pode usar `underscore_form` como nome das suas **migrations**. Ex.: `create_products`.

Novo na versão cakephp/migrations: 1.5.2

A partir da versão 1.5.2 do [plugin migrations](#)¹⁶², o nome dos arquivos de migrações são colocados automaticamente no padrão **camel case**. Esta versão do plugin está disponível apenas a partir da versão 3.1 do CakePHP. Antes disto, o padrão de nomes do plugin migrations utilizava a nomenclatura baseada em **underlines**, ex.: `20160121164955_create_products.php`.

Aviso: O nome das migrações são usados como nomes de classe, e podem colidir com outras migrações se o nome das classes não forem únicos. Neste caso, pode ser necessário sobrescrever manualmente os nomes mais tarde ou simplesmente mudar os nomes que você está especificando.

Definição de colunas

Quando utilizar colunas na linha de comando, pode ser útil lembrar que eles seguem o seguinte padrão:

```
fieldName:fieldType[length]:indexType:indexName
```

Por exemplo, veja formas válidas de especificar um campo de e-mail:

- `email:string:unique`
- `email:string:unique:EMAIL_INDEX`
- `email:string[120]:unique:EMAIL_INDEX`

O parâmetro `length` para o `fieldType` é opcional e deve sempre ser escrito entre colchetes

Os campos `created` e `modified` serão automaticamente definidos como `datetime`.

Os tipos de campos são genericamente disponibilizados pela biblioteca `Phinx`. Eles podem ser:

- `string`
- `text`
- `integer`
- `biginteger`
- `float`
- `decimal`
- `datetime`

¹⁶² <https://github.com/cakephp/migrations/>

- timestamp
- time
- date
- binary
- boolean
- uuid

Há algumas heurísticas para a escolha de tipos de campos que não são especificados ou são definidos com valor inválido. O tipo de campo padrão é string;

- id: integer
- created, modified, updated: datetime

Criando uma tabela

Você pode utilizar o bake para criar uma tabela:

```
$ bin/cake bake migration CreateProducts name:string description:text created modified
```

A linha de comando acima irá gerar um arquivo de migração parecido com este:

```
<?php
use Migrations\AbstractMigration;

class CreateProducts extends AbstractMigration
{
    /**
     * Change Method.
     *
     * More information on this method is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-change-method
     * @return void
     */
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('name', 'string', [
            'default' => null,
            'limit' => 255,
            'null' => false,
        ]);
        $table->addColumn('description', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => false,
        ]);
        $table->create();
    }
}
```

```
}  
}
```

Adicionando colunas a uma tabela existente

Se o nome da migração na linha de comando estiver na forma “AddXXXToYYY” e for seguido por uma lista de nomes de colunas e tipos, então o arquivo de migração com o código para criar as colunas será gerado:

```
$ bin/cake bake migration AddPriceToProducts price:decimal
```

A linha de comando acima irá gerar um arquivo com o seguinte conteúdo:

```
<?php  
use Migrations\AbstractMigration;  
  
class AddPriceToProducts extends AbstractMigration  
{  
    public function change()  
    {  
        $table = $this->table('products');  
        $table->addColumn('price', 'decimal')  
            ->update();  
    }  
}
```

Adicionando uma coluna como índice a uma tabela

Também é possível adicionar índices a colunas:

```
$ bin/cake bake migration AddNameIndexToProducts name:string:index
```

irá gerar:

```
<?php  
use Migrations\AbstractMigration;  
  
class AddNameIndexToProducts extends AbstractMigration  
{  
    public function change()  
    {  
        $table = $this->table('products');  
        $table->addColumn('name', 'string')  
            ->addIndex(['name'])  
            ->update();  
    }  
}
```

Especificando o tamanho do campo

Novo na versão cakephp/migrations: 1.4

Se você precisar especificar o tamanho do campo, você pode fazer isto entre colchetes logo após o tipo do campo, ex.:

```
$ bin/cake bake migration AddFullDescriptionToProducts full_description:string[60]
```

Executar o comando acima irá gerar:

```
<?php
use Migrations\AbstractMigration;

class AddFullDescriptionToProducts extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products');
        $table->addColumn('full_description', 'string', [
            'default' => null,
            'limit' => 60,
            'null' => false,
        ])
        ->update();
    }
}
```

Se o tamanho não for especificado, os seguintes padrões serão utilizados:

- string: 255
- integer: 11
- biginteger: 20

Removendo uma coluna de uma tabela

Da mesma forma, você pode gerar uma migração para remover uma coluna utilizando a linha de comando, se o nome da migração estiver na forma “RemoveXXXFromYYY”:

```
$ bin/cake bake migration RemovePriceFromProducts price
```

Cria o arquivo:

```
<?php
use Migrations\AbstractMigration;

class RemovePriceFromProducts extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products');
        $table->removeColumn('price');
    }
}
```

Gerando migrações a partir de uma base de dados existente

Se você está trabalhando com um banco de dados pré-existente e quer começar a usar migrações, ou para versionar o schema inicial da base de dados da sua aplicação, você pode executar o comando `migration_snapshot`:

```
$ bin/cake bake migration_snapshot Initial
```

Isto irá gerar um arquivo de migração chamado **YYYYMMDDHHMMSS_Initial.php** contendo todas as instruções CREATE para todas as tabelas no seu banco de dados.

Por padrão, o snapshot será criado a partir da conexão default definida na configuração. Se você precisar fazer o bake de um snapshot de uma fonte de dados diferente, você pode utilizar a opção `--connection`:

```
$ bin/cake bake migration_snapshot Initial --connection my_other_connection
```

Você também pode definir que o snapshot inclua apenas as tabelas para as quais você tenha definido models correspondentes, utilizando a flag `require-table`:

```
$ bin/cake bake migration_snapshot Initial --require-table
```

Quando utilizar a flag `--require-table`, o shell irá olhar através das classes do diretório Table da sua aplicação e apenas irá adicionar no snapshot as tabelas lá definidas.

A mesma lógica será aplicada implicitamente se você quiser fazer o bake de um snapshot para um plugin. Para fazer isso, você precisa usar a opção `--plugin`, veja a seguir:

```
$ bin/cake bake migration_snapshot Initial --plugin MyPlugin
```

Apenas as tabelas que tiverem um objeto Table definido serão adicionadas ao snapshot do seu plugin.

Nota: Quando fizer o bake de um snapshot para um plugin, os arquivos de migrações serão criados no diretório **config/Migrations** do seu plugin.

Fique atento que quando você faz o bake de um snapshot, ele é automaticamente adicionado ao log do phinx como migrado.

Os Comandos

migrate : Aplicando migrações

Depois de ter gerado ou escrito seu arquivo de migração, você precisa executar o seguinte comando para aplicar as mudanças a sua base de dados:

```
# Executa todas as migrações
$ bin/cake migrations migrate

# Execute uma migração específica utilizando a opção ``--target`` ou ``-t``
# O valor é um timestamp que serve como prefixo para cada migração::
$ bin/cake migrations migrate -t 20150103081132

# Por padrão, as migrações ficam no diretório **config/Migrations**. Você
# pode especificar um diretório utilizando a opção ``--source`` ou ``-s``.
# O comando abaixo executa as migrações no diretório **config/Alternate**
$ bin/cake migrations migrate -s Alternate

# Você pode executar as migrações de uma conexão diferente da ``default``
# utilizando a opção ``--connection`` ou ``-c``.
$ bin/cake migrations migrate -c my_custom_connection
```

```
# Migrações também podem ser executadas para plugins. Simplesmente utilize
# a opção ``--plugin`` ou ``-p``
$ bin/cake migrations migrate -p MyAwesomePlugin
```

rollback : Revertendo migrações

O comando `rollback` é utilizado para desfazer migrações realizadas anteriormente pelo plugin Migrations. É o inverso do comando `migrate`:

```
# Você pode desfazer uma migração anterior utilizando o
# comando ``rollback``:
$ bin/cake migrations rollback

# Você também pode passar a versão da migração para voltar
# para uma versão específica:
$ bin/cake migrations rollback -t 20150103081132
```

Você também pode utilizar as opções `--source`, `--connection` e `--plugin` exatamente como no comando `migrate`.

status : Status da migração

O comando `status` exibe uma lista de todas as migrações juntamente com seu status. Você pode utilizar este comando para ver quais migrações foram executadas:

```
$ bin/cake migrations status
```

Você também pode ver os resultados como JSON utilizando a opção `--format` (ou `-f`):

```
$ bin/cake migrations status --format json
```

Você também pode utilizar as opções `--source`, `--connection` e `--plugin` exatamente como no comando `migrate`.

mark_migrated : Marcando uma migração como migrada

Novo na versão 1.4.0.

Algumas vezes pode ser útil marcar uma lista de migrações como migrada sem efetivamente executá-las. Para fazer isto, você pode usar o comando `mark_migrated`. O comando é bastante semelhante aos outros comandos.

Você pode marcar todas as migrações como migradas utilizando este comando:

```
$ bin/cake migrations mark_migrated
```

Você também pode marcar todas as migrações de uma versão específica utilizando a opção `--target`:

```
$ bin/cake migrations mark_migrated --target=20151016204000
```

Se você não quer marcar a migração alvo como migrada durante o processo, você pode utilizar a opção `--exclude`:

```
$ bin/cake migrations mark_migrated --target=20151016204000 --exclude
```

Finalmente, se você deseja marcar somente a migração alvo como migrada, você pode utilizar a opção `--only`:

```
$ bin/cake migrations mark_migrated --target=20151016204000 --only
```

Você também pode utilizar as opções `--source`, `--connection` e `--plugin` exatamente como no comando `migrate`.

Nota: Quando você criar um snapshot utilizando o `bake` com o comando `cake bake migration_snapshot`, a migração criada será automaticamente marcada como migrada.

Obsoleto desde a versão 1.4.0: A seguinte maneira de utilizar o comando foi depreciada. Use somente se você estiver utilizando uma versão do plugin inferior a 1.4.0.

Este comando espera um número de versão de migração como argumento:

```
$ bin/cake migrations mark_migrated
```

Se você deseja marcar todas as migrações como migradas, você pode utilizar o valor especial `all`. Se você o utilizar, ele irá marcar todas as migrações como migradas:

```
$ bin/cake migrations mark_migrated all
```

seed : Populando seu banco de dados

A partir da versão 1.5.5, você pode usar a **shell** de `migrations` para popular seu banco de dados. Essa função é oferecida graças ao [recurso de seed da biblioteca Phinx](http://docs.phinx.org/en/latest/seeding.html)¹⁶³. Por padrão, arquivos **seed** ficarão no diretório `config/Seeds` de sua aplicação. Por favor, tenha certeza de seguir as [instruções do Phinx para construir seus arquivos de seed](http://docs.phinx.org/en/latest/seeding.html#creating-a-new-seed-class)¹⁶⁴.

Assim como nos **migrations**, uma interface do ``bake` é oferecida para gerar arquivos de **seed**:

```
# This will create a ArticlesSeed.php file in the directory config/Seeds of your
↪ application
# By default, the table the seed will try to alter is the "tableized" version of the
↪ seed filename
$ bin/cake bake seed Articles

# You specify the name of the table the seed files will alter by using the `--
↪ table` option
$ bin/cake bake seed Articles --table my_articles_table

# You can specify a plugin to bake into
$ bin/cake bake seed Articles --plugin PluginName

# You can specify an alternative connection when generating a seeder.
$ bin/cake bake seed Articles --connection connection
```

Para popular seu banco de dados, você pode usar o subcomando `seed`:

¹⁶³ <http://docs.phinx.org/en/latest/seeding.html>

¹⁶⁴ <http://docs.phinx.org/en/latest/seeding.html#creating-a-new-seed-class>

```
# Without parameters, the seed subcommand will run all available seeders
# in the target directory, in alphabetical order.
$ bin/cake migrations seed

# You can specify only one seeder to be run using the `--seed` option
$ bin/cake migrations seed --seed ArticlesSeed

# You can run seeders from an alternative directory
$ bin/cake migrations seed --source AlternativeSeeds

# You can run seeders from a plugin
$ bin/cake migrations seed --plugin PluginName

# You can run seeders from a specific connection
$ bin/cake migrations seed --connection connection
```

Esteja ciente que, ao oposto das **migrations**, **seeders** não são versionados, o que significa que o mesmo **seeder** pode ser aplicado diversas vezes.

Usando migrations em plugins

Plugins também podem oferecer **migrations**. Isso faz com que **plugins** que são planejados para serem distribuídos tornem-se muito mais práticos e fáceis de instalar. Todos os comandos do plugin **Migrations** suportam a opção `--plugin` ou `-p`, que por sua vez vai delegar a execução da tarefa ao escopo relativo a um determinado **plugin**:

```
$ bin/cake migrations status -p PluginName
$ bin/cake migrations migrate -p PluginName
```

Executando migrations em ambientes fora da linha de comando

Novo na versão cakephp/migrations: 1.2.0

Desde o lançamento da versão 1.2 do plugin, você pode executar **migrations** fora da linha de comando, diretamente de uma aplicação, ao usar a nova classe `Migrations`. Isso pode ser muito útil caso você esteja desenvolvendo um instalador de **plugins** para um CMS, para exemplificar.

A classe `Migrations` permite que você execute os seguintes comandos disponíveis na **shell**:

- `migrate`
- `rollback`
- `markMigrated`
- `status`
- `seed`

Cada um desses comandos tem um método definido na classe `Migrations`.

Veja como usá-la:

```
use Migrations\Migrations;
```

```
$migrations = new Migrations();

// Retornará um array de todos migrations e seus status
$status = $migrations->status();

// Retornará true se bem sucedido. Se um erro ocorrer, uma exceção será lançada
$migrate = $migrations->migrate();

// Retornará true se bem sucedido. Se um erro ocorrer, uma exceção será lançada
$rollback = $migrations->rollback();

// Retornará true se bem sucedido. Se um erro ocorrer, uma exceção será lançada
$markMigrated = $migrations->markMigrated(20150804222900);

// Retornará true se bem sucedido. Se um erro ocorrer, uma exceção será lançada
$seeded = $migrations->seed();
```

Os métodos aceitam um **array** de parâmetros que devem combinar com as opções dos comandos:

```
use Migrations\Migrations;

$migrations = new Migrations();

// Retornará um array de todos migrations e seus status
$status = $migrations->status(['connection' => 'custom', 'source' =>
    ↪ 'MyMigrationsFolder']);
```

Você pode passar qualquer opção que esteja disponível pelos comandos **shell**. A única exceção é o comando `markMigrated` que espera um número de versão a ser marcado como migrado, como primeiro argumento. Passe o **array** de parâmetros como segundo argumento nesse caso.

Opcionalmente, você pode passar esses parâmetros pelo construtor da classe. Eles serão usados como padrão evitando que você tenha que passá-los em cada chamada do método:

```
use Migrations\Migrations;

$migrations = new Migrations(['connection' => 'custom', 'source' =>
    ↪ 'MyMigrationsFolder']);

// Todas as chamadas de métodos serão executadas usando os parâmetros passados pelo
    ↪ construtor da classe
$status = $migrations->status();
$migrate = $migrations->migrate();
```

Se você precisar sobrescrever um ou mais parâmetros definidos previamente, você pode passá-los para um método:

```
use Migrations\Migrations;

$migrations = new Migrations(['connection' => 'custom', 'source' =>
    ↪ 'MyMigrationsFolder']);

// Essa chamada será feita com a conexão "custom"
$status = $migrations->status();
// Essa chamada será feita com a conexão "default"
$migrate = $migrations->migrate(['connection' => 'default']);
```


Dicas e truques

Criando chaves primárias customizadas

Se você precisa evitar a criação automática da chave primária `id` ao adicionanr novas tabelas ao banco de dados, é possível usar o segundo argumento do método `table()`:

```
<?php
use Migrations\AbstractMigration;

class CreateProductsTable extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('products', ['id' => false, 'primary_key' => ['id']]);
        $table
            ->addColumn('id', 'uuid')
            ->addColumn('name', 'string')
            ->addColumn('description', 'text')
            ->create();
    }
}
```

O código acima vai criar uma coluna `CHAR(36) id` que também é a chave primária.

Nota: Ao especificar chaves primárias customizadas pela linha de comando, você deve apontá-las como chave primária no campo `id`, caso contrário você pode receber um erro apontando campos duplicados, i.e.:

```
$ bin/cake bake migration CreateProducts id:uuid:primary name:string description:text_
↳ created modified
```

Adicionalmente, desde a versão 1.3, uma novo meio de lidar com chaves primárias foi introduzido. Para tal, sua classe de migração deve estender a nova classe `Migrations\AbstractMigration`.

Você pode especificar uma propriedade `autoId` na sua classe e defini-la como `false`, o quê desabilitará a geração automática da coluna `id`. Você vai precisar criar manualmente a coluna que será usada como chave primária e adicioná-la à declaração da tabela:

```
<?php
use Migrations\AbstractMigration;

class CreateProductsTable extends AbstractMigration
{
    public $autoId = false;

    public function up()
    {
        $table = $this->table('products');
        $table
            ->addColumn('id', 'integer', [
                'autoIncrement' => true,
                'limit' => 11
            ])
            ->addPrimaryKey('id')
    }
}
```

```
->addColumn('name', 'string')
->addColumn('description', 'text')
->create();
}
}
```

Comparado ao método apresentado anteriormente de lidar com chaves primárias, nesse método, temos a possibilidade de ter maior controle sobre as definições da coluna da chave primária: unsigned, limit, comentários, etc.

Todas as migrations e snapshots criadas pelo bake vão usar essa nova forma quando necessário.

Aviso: Lidar com chaves primárias só é possível no momento de criação de tabelas. Isso é devido a algumas limitações de alguns servidores de banco de dados que o plugin suporta.

Colações

Se você precisar criar uma tabela com colação diferente do padrão do banco de dados, você pode defini-la pelo método `table()`, como uma opção:

```
<?php
use Migrations\AbstractMigration;

class CreateCategoriesTable extends AbstractMigration
{
    public function change()
    {
        $table = $this
            ->table('categories', [
                'collation' => 'latin1_german1_ci'
            ])
            ->addColumn('title', 'string', [
                'default' => null,
                'limit' => 255,
                'null' => false,
            ])
            ->create();
    }
}
```

Note que isso só pode ser feito na criação da tabela : não há atualmente uma forma de adicionar uma coluna a uma tabela existente com uma colação diferente do padrão da tabela, ou mesmo do banco de dados. Apenas MySQL e SqlServer suportam essa chave de configuração.

Atualizando nome de colunas e usando objetos de tabela

Se você usa um objeto ORM Table do CakePHP para manipular valores do seu banco de dados, renomeando ou removendo uma coluna, certifique-se de criar uma nova instância do seu objeto depois da chamada do `update()`. O registro do objeto é limpo depois da chamada do `update()` para atualizar o **schema** que é refletido e armazenado no objeto Table paralelo à instanciação.

Migrations e Deployment

Se você usa o plugin ao fazer o **deploy** de sua aplicação, garanta que o cache ORM seja limpo para renovar os metadados das colunas de suas tabelas. Caso contrário, você pode acabar recebendo erros relativos a colunas inexistentes ao criar operações nessas mesmas colunas. O **core** do CakePHP possui uma *ORM Cache Shell* que você pode usar para realizar essas operação:

```
$ bin/cake orm_cache clear
```

Leia a seção *ORM Cache Shell* do cookbook se você quiser conhecer mais sobre essa **shell**.

Apêndices

Os apêndices contêm informações sobre os novos recursos introduzidos em cada versão e a forma de executar a migração entre versões.

Guia de Migração para a versão 3.x

3.x Migration Guide

Migration guides contain information regarding the new features introduced in each version and the migration path between versions.

3.5 Migration Guide

3.5 Migration Guide

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)¹⁶⁵ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

3.4 Migration Guide

3.4 Migration Guide

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintase a vontade para nos enviar um *pull request* para o [Github](#)¹⁶⁶ ou use o botão **IMPROVE THIS DOC**

¹⁶⁵ <https://github.com/cakephp/docs>

¹⁶⁶ <https://github.com/cakephp/docs>

para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

3.3 Migration Guide

3.3 Migration Guide

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁶⁷ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

3.2 Migration Guide

3.2 Guia de migração

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁶⁸ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

3.1 Migration Guide

3.1 Guia de migração

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sinta-se a vontade para nos enviar um *pull request* para o [Github](#)¹⁶⁹ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

¹⁶⁷ <https://github.com/cakephp/docs>

¹⁶⁸ <https://github.com/cakephp/docs>

¹⁶⁹ <https://github.com/cakephp/docs>

3.0 Migration Guide

Guia de atualização para o novo ORM

Nota: Atualmente, a documentação desta página não é suportada em português.

Por favor, sintá-se a vontade para nos enviar um *pull request* para o [Github](https://github.com/cakephp/docs)¹⁷⁰ ou use o botão **IMPROVE THIS DOC** para propor suas mudanças diretamente.

Você pode consultar a versão em inglês deste tópico através do seletor de idiomas localizado ao lado direito do campo de buscas da documentação.

Informações Gerais

Processo de desenvolvimento no CakePHP

Aqui tentamos explicar o processo utilizado no desenvolvimento com o framework CakePHP. Nós dependemos fortemente da interação por tickets e no canal do IRC. O IRC é o melhor lugar para encontrar membros do *time de desenvolvimento*¹⁷¹ e discutir idéias, o último código e fazer comentários gerais. Se algo mais formal tem que ser proposto ou existe um problema com uma versão, o sistema de tickets é o melhor lugar para compartilhar seus pensamentos.

Nós atualmente mantemos 4 versões do CakePHP.

- **versões tageadas** : Versões tageadas são destinadas para produção onde uma estabilidade maior é mais importante do que funcionalidades. Questões sobre versões tageadas serão resolvidas no branch relacionado e serão parte do próximo release.
- **branch principal** : Esses branches são onde todas as correções são fundidas. Versões estáveis são rotuladas a partir desses branches. `master` é o principal branch para a versão atual. `2.x` é o branch de manutenção para a versão 2.x. Se você está usando versões estáveis e precisa de correções que não chegaram em uma versão tageada olhe aqui.
- **desenvolvimento** : O branch de desenvolvimento contém sempre as últimas correções e funcionalidades. Eles são nomeados pela versão a qual se destinam, ex: *3.next*. Uma vez que estas branches estão estáveis elas são fundidas na branch principal da versão.
- **branches de funcionalidades** : Branches de funcionalidade contém trabalhos que estão sendo desenvolvidos ou possivelmente instáveis e são recomendadas apenas para usuários avançados interessados e dispostos a contribuir com a comunidade. Branches de funcionalidade são nomeadas pela seguinte convenção *versão-funcionalidade*. Um exemplo seria *3.3-router* Que conteria novas funcionalidades para o Router na 3.3

Esperamos que isso te ajudará a entender que versão é correta pra você. Uma vez que escolhida a versão você pode se sentir compelido a reportar um erro ou fazer comentários gerais no código.

- Se você está usando uma versão estável ou de manutenção, por favor envie tickets ou discuta conosco no IRC.
- Se você está usando uma branch de desenvolvimento ou funcionalidade, o primeiro lugar para ir é o IRC. Se você tem um comentário e não consegue entrar no IRC depois de um ou dois dias, envie um ticket.

Se você encontrar um problema, a melhor resposta é escrever um teste. O melhor conselho que podemos oferecer em escrever testes é olhar nos que estão no núcleo do projeto.

¹⁷⁰ <https://github.com/cakephp/docs>

¹⁷¹ <https://github.com/cakephp?tab=members>

E sempre, se você tiver alguma questão ou comentários, nos visite no #cakephp no irc.freenode.net

Glossário

routing array Uma série de atributos que são passados para `Router::url()`. Eles normalmente parecem:

```
['controller' => 'Posts', 'action' => 'view', 5]
```

HTML attributes Uma série de arrays key => values que são compostos em atributos HTML. Por Exemplo:

```
// Tendo isso
['class' => 'my-class', 'target' => '_blank']

// Geraria isto
class="my-class" target="_blank"
```

Se uma opção pode ser minimizada ou aceitar seu nome como valor, então `true` pode ser usado:

```
// Tendo isso
['checked' => true]

// Geraria isto
checked="checked"
```

sintaxe plugin A sintaxe do plugin refere-se ao nome da classe separada por pontos que indica classes:

```
// O plugin "DebugKit", e o nome da "Toolbar".
'DebugKit.Toolbar'

// O plugin "AcmeCorp/Tools", e o nome da class "Toolbar".
'AcmeCorp/Tools.Toolbar'
```

dot notation A notação de ponto define um caminho do array, separando níveis aninhados com `.`. Por exemplo:

```
Cache.default.engine
```

Geraria o seguinte valor:

```
[
    'Cache' => [
        'default' => [
            'engine' => 'File'
        ]
    ]
]
```

CSRF Cross Site Request Forgery. Impede ataques de repetição, envios duplos e solicitações forjadas de outros domínios.

CDN Content Delivery Network. Um fornecedor de código de terceiros que você pode pagar para ajudar a distribuir seu conteúdo para centros de dados em todo o mundo. Isso ajuda a colocar seus ativos estáticos mais próximos dos usuários distribuídos geograficamente.

routes.php O arquivo `config` diretório que contém configuração de roteamento. Este arquivo está incluído antes de cada solicitação ser processada. Ele deve conectar todas as rotas que seu aplicativo precisa para que as solicitações possam ser encaminhadas para a ação correta do controlador.

DRY Não se repita. É um princípio de desenvolvimento de software destinado a reduzir a repetição de informações de todos os tipos. No CakePHP DRY é usado para permitir codificar coisas uma vez e reutilizá-las em toda a sua aplicação.

PaaS Plataforma como um serviço. A plataforma como um provedor de serviços fornecerá recursos baseados em nuvem de hospedagem, banco de dados e armazenamento em cache. Alguns provedores populares incluem Heroku, EngineYard e PagodaBox.

DSN Data Source Name. Um formato de sequência de conexão que é formado como um URI. O CakePHP suporta DSN para conexões Cache, Database, Log e Email.

PHP Namespace Index

C

- Cake\Console, [287](#)
- Cake\Controller, [163](#)
- Cake\Core, [126](#)
- Cake\Core\Configure, [131](#)
- Cake\Core\Configure\Engine, [131](#)
- Cake\Database, [203](#)
- Cake\Database\Schema, [275](#)
- Cake\Datasource, [203](#)
- Cake>Error, [306](#)
- Cake\Form, [331](#)
- Cake\Mailer, [311](#)
- Cake\ORM, [244](#)
- Cake\Routing, [137](#)
- Cake\Utility, [361](#)
- Cake\View, [175](#)
- Cake\View\Helper, [195](#)

Symbols

() (Cake\Console\ method), **291, 292**
() (método), **214**
:action, **138**
:controller, **138**
:plugin, **138**
\$this->request, **161**
__() (global function), **395**
__d() (global function), **395**
__dn() (global function), **395**
__dx() (global function), **395**
__dxn() (global function), **395**
__n() (global function), **396**
__x() (global function), **396**
__xn() (global function), **396**

A

addArgument() (Cake\Console\ConsoleOptionParser method), **296**
addArguments() (Cake\Console\ConsoleOptionParser method), **296**
addOption() (Cake\Console\ConsoleOptionParser method), **297**
addOptions() (Cake\Console\ConsoleOptionParser method), **297**
addSubcommand() (Cake\Console\ConsoleOptionParser method), **298**
admin routing, **145**
afterFilter() (Cake\Controller\Controller method), **170**
APP (global constant), **397**
app.php, **123**
app.php.default, **123**
APP_DIR (global constant), **397**
attachments() (Cake\Mailer\Email method), **316**

B

beforeFilter() (Cake\Controller\Controller method), **170**

beforeRender() (Cake\Controller\Controller method), **170**
BreadcrumbsHelper (classe em Cake\View\Helper), **192**
breakpoint() (global function), **305**
buildFromArray() (Cake\Console\ConsoleOptionParser method), **299**

C

CACHE (global constant), **397**
cache() (Cake\View\View method), **186**
CAKE (global constant), **397**
CAKE_CORE_INCLUDE_PATH (global constant), **397**
Cake\Console (namespace), **287**
Cake\Controller (namespace), **163**
Cake\Core (namespace), **126**
Cake\Core\Configure (namespace), **131**
Cake\Core\Configure\Engine (namespace), **131**
Cake\Database (namespace), **203**
Cake\Database\Schema (namespace), **275**
Cake\Datasource (namespace), **203**
Cake>Error (namespace), **306**
Cake\Form (namespace), **331**
Cake\Mailer (namespace), **311**
Cake\ORM (namespace), **215, 223, 244, 266**
Cake\Routing (namespace), **137**
Cake\Utility (namespace), **361, 381**
Cake\View (namespace), **175**
Cake\View\Helper (namespace), **192–195**
camelize() (Cake\Utility\Inflector method), **382**
CDN, **424**
check() (Cake\Core\Configure method), **127**
classify() (Cake\Utility\Inflector method), **383**
Collection (classe em Cake\Database\Schema), **279**
collection() (global function), **396**
components (Cake\Controller\Controller property), **169**
CONFIG (global constant), **397**
config() (Cake\Core\Configure method), **128**
ConfigEngineInterface (interface in
Cake\Core\Configure), **131**

configTransport() (Cake\Mailer\Email method), [313](#)
configuration, [123](#)
Configure (classe em Cake\Core), [126](#)
connect() (Cake\Routing\Router method), [139](#)
Connection (classe em Cake\Database), [208](#)
ConnectionManager (classe em Cake\Datasource), [203](#)
ConsoleOptionParser (classe em Cake\Console), [294](#)
consume() (Cake\Core\Configure method), [128](#)
Controller (classe em Cake\Controller), [163](#)
CORE_PATH (global constant), [397](#)
CSRF, [424](#)

D

dasherize() (Cake\Utility\Inflector method), [383](#)
DAY (global constant), [398](#)
debug() (global function), [396](#)
Debugger (classe em Cake/Error), [306](#)
decrypt() (Cake\Utility\Security method), [361](#)
defaultRouteClass() (Cake\Routing\Router method), [158](#)
delete() (Cake\Core\Configure method), [128](#)
delete() (Cake\ORM\Table method), [266](#)
deleteAll() (Cake\ORM\Table method), [267](#)
deleteOrFail() (Cake\ORM\Table method), [267](#)
description() (Cake\Console\ConsoleOptionParser method), [295](#)
dirty() (Cake\ORM\Entity method), [218](#)
doc (papêl), [95](#)
dot notation, [424](#)
drop() (Cake\Core\Configure method), [128](#)
dropTransport() (Cake\Mailer\Email method), [314](#)
DRY, [425](#)
DS (global constant), [397](#)
DSN, [425](#)
dump() (Cake\Core\Configure method), [129](#)
dump() (Cake\Core\Configure\ConfigEngineInterface method), [131](#)
dump() (Cake/Error\Debugger method), [306](#)

E

element() (Cake\View\View method), [183](#)
Email (classe em Cake\Mailer), [311](#)
encrypt() (Cake\Utility\Security method), [361](#)
Entity (classe em Cake\ORM), [215](#)
env() (global function), [396](#)
epilog() (Cake\Console\ConsoleOptionParser method), [296](#)
errors() (Cake\ORM\Entity method), [219](#)
excerpt() (Cake/Error\Debugger method), [307](#)
execute() (Cake\Database\Connection method), [208](#)
extensions() (Cake\Routing\Router method), [149](#)

F

fallbacks() (Cake\Routing\Router method), [158](#)
file extensions, [149](#)

find() (Cake\ORM\Table method), [225](#)
FlashHelper (classe em Cake\View\Helper), [192](#)
Form (classe em Cake\Form), [331](#)
FormHelper (classe em Cake\View\Helper), [193](#)

G

get() (Cake\Datasource\ConnectionManager method), [203](#)
get() (Cake\ORM\Entity method), [217](#)
get() (Cake\ORM\Table method), [224](#)
getType() (Cake/Error\Debugger method), [307](#)
greedy star, [138](#)

H

h() (global function), [396](#)
hash() (Cake\Utility\Security method), [362](#)
helpers (Cake\Controller\Controller property), [170](#)
HOUR (global constant), [398](#)
HTML attributes, [424](#)
HtmlHelper (classe em Cake\View\Helper), [193](#)
humanize() (Cake\Utility\Inflector method), [382](#)

I

Inflector (classe em Cake\Utility), [381](#)
IniConfig (classe em Cake\Core\Configure\Engine), [131](#)
initialize() (Cake\Console\ConsoleOptionParser method), [302](#)

J

JsonConfig (classe em Cake\Core\Configure\Engine), [132](#)

L

load() (Cake\Core\Configure method), [129](#)
loadComponent() (Cake\Controller\Controller method), [169](#)
loadModel() (Cake\Controller\Controller method), [168](#)
log() (Cake/Error\Debugger method), [306](#)
LOGS (global constant), [397](#)

M

map() (Cake\Database\Type method), [204](#)
MINUTE (global constant), [398](#)
MONTH (global constant), [398](#)

N

namespaceSplit() (global function), [397](#)
newQuery() (Cake\Database\Connection method), [208](#)
NumberHelper (classe em Cake\View\Helper), [193](#)

P

PaaS, [425](#)
paginate() (Cake\Controller\Controller method), [169](#)
PaginatorHelper (classe em Cake\View\Helper), [193](#)

passed arguments, [155](#)
 php:attr (diretiva), [96](#)
 php:attr (papel), [97](#)
 php:class (diretiva), [96](#)
 php:class (papel), [97](#)
 php:const (diretiva), [96](#)
 php:const (papel), [97](#)
 php:exc (papel), [97](#)
 php:exception (diretiva), [96](#)
 php:func (papel), [97](#)
 php:function (diretiva), [96](#)
 php:global (diretiva), [96](#)
 php:global (papel), [97](#)
 php:meth (papel), [97](#)
 php:method (diretiva), [96](#)
 php:staticmethod (diretiva), [96](#)
 PhpConfig (classe em Cake\Core\ConfigEngine), [131](#)
 pj() (global function), [396](#)
 plugin routing, [147](#)
 plugin() (Cake\Routing\Router method), [147](#)
 pluginSplit() (global function), [396](#)
 pluralize() (Cake\Utility\Inflector method), [382](#)
 pr() (global function), [396](#)
 prefix routing, [145](#)
 prefix() (Cake\Routing\Router method), [146](#)

Q

query() (Cake\Database\Connection method), [208](#)

R

randomBytes() (Cake\Utility\Security method), [363](#)
 read() (Cake\Core\Config method), [127](#)
 read() (Cake\Core\ConfigEngineInterface method), [131](#)
 readOrFail() (Cake\Core\Config method), [127](#)
 redirect() (Cake\Controller\Controller method), [167](#)
 ref (papel), [95](#)
 render() (Cake\Controller\Controller method), [166](#)
 restore() (Cake\Core\Config method), [130](#)
 RFC
 RFC 2606, [110](#)
 ROOT (global constant), [397](#)
 Router (classe em Cake\Routing), [137](#)
 routes.php, [137](#), [424](#)
 routing array, [424](#)
 RssHelper (classe em Cake\View\Helper), [194](#)
 rules() (Cake\Utility\Inflector method), [384](#)

S

save() (Cake\ORM\Table method), [256](#)
 saveMany() (Cake\ORM\Table method), [265](#)
 saveOrFail() (Cake\ORM\Table method), [264](#)
 SECOND (global constant), [398](#)

Security (classe em Cake\Utility), [361](#)
 SessionHelper (classe em Cake\View\Helper), [194](#)
 set() (Cake\Controller\Controller method), [165](#)
 set() (Cake\ORM\Entity method), [217](#)
 set() (Cake\View\View method), [177](#)
 setAction() (Cake\Controller\Controller method), [168](#)
 singularize() (Cake\Utility\Inflector method), [382](#)
 sintaxe plugin, [424](#)
 slug() (Cake\Utility\Inflector method), [383](#)
 stackTrace() (global function), [305](#)
 startup() (Cake\Console\ConsoleOptionParser method), [302](#)
 store() (Cake\Core\Config method), [129](#)

T

Table (classe em Cake\ORM), [223](#)
 tableize() (Cake\Utility\Inflector method), [383](#)
 TableSchema (classe em Cake\Database\Schema), [275](#)
 TESTS (global constant), [397](#)
 TextHelper (classe em Cake\View\Helper), [194](#)
 TIME_START (global constant), [397](#)
 TimeHelper (classe em Cake\View\Helper), [195](#)
 TMP (global constant), [397](#)
 trace() (Cake>Error\Debugger method), [306](#)
 trailing star, [138](#)
 transactional() (Cake\Database\Connection method), [209](#)
 Type (classe em Cake\Database), [203](#)

U

underscore() (Cake\Utility\Inflector method), [382](#)
 updateAll() (Cake\ORM\Table method), [265](#)
 url() (Cake\Routing\Router method), [156](#)
 UrlHelper (classe em Cake\View\Helper), [195](#)

V

variable() (Cake\Utility\Inflector method), [383](#)
 vendor/cakephp-plugins.php, [345](#)
 View (classe em Cake\View), [175](#)

W

WEEK (global constant), [398](#)
 write() (Cake\Core\Config method), [126](#)
 WWW_ROOT (global constant), [397](#)

Y

YEAR (global constant), [398](#)