CakePHP Book

Versão 5.x

Cake Software Foundation

25 Oct, 2024

Conteúdo

CAPÍTULO 1

CakePHP num piscar de olhos

O CakePHP é desenvolvido para tornar tarefas rotineiras do desenvolvimento web mais simples e fáceis. Ao fornecer uma caixa de ferramentas completa para você começar, as várias partes do CakePHP funcionam bem juntas ou separadamente.

O objetivo desta apresentação é introduzir os conceitos gerais presentes no CakePHP e lhe dar uma rápida visão geral de como esses conceitos são implementados. Se você está ansioso 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, arquivos, banco de dados e outras convenções. Apesar das convenções levarem algum tempo para serem assimiladas, ao seguí-las você evita configurações desnecessárias e cria uma estrutura de aplicação uniforme, que faz trabalhar com vários projetos uma tarefa suave. O *capítulo de convenções* explica as várias convenções que o CakePHP utiliza.

A Camada Model

A camada Model representa a parte da sua aplicação que implementa a lógica de negócios. 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 cuidar 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;

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->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 que ainda não foram definidas.

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

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->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 'element' para cada usuário.
<?php foreach ($users as $user): ?>
    <div class="user">
        <?= $this->element('user', ['user' => $user]) ?>
    </div>
<?php endforeach; ?>
```

A camada View fornece vários pontos de extensão, como *View Templates*, *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 a 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.

Um controller pode ser visto como um gerente que certifica-se de 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, seleciona 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);
}
```

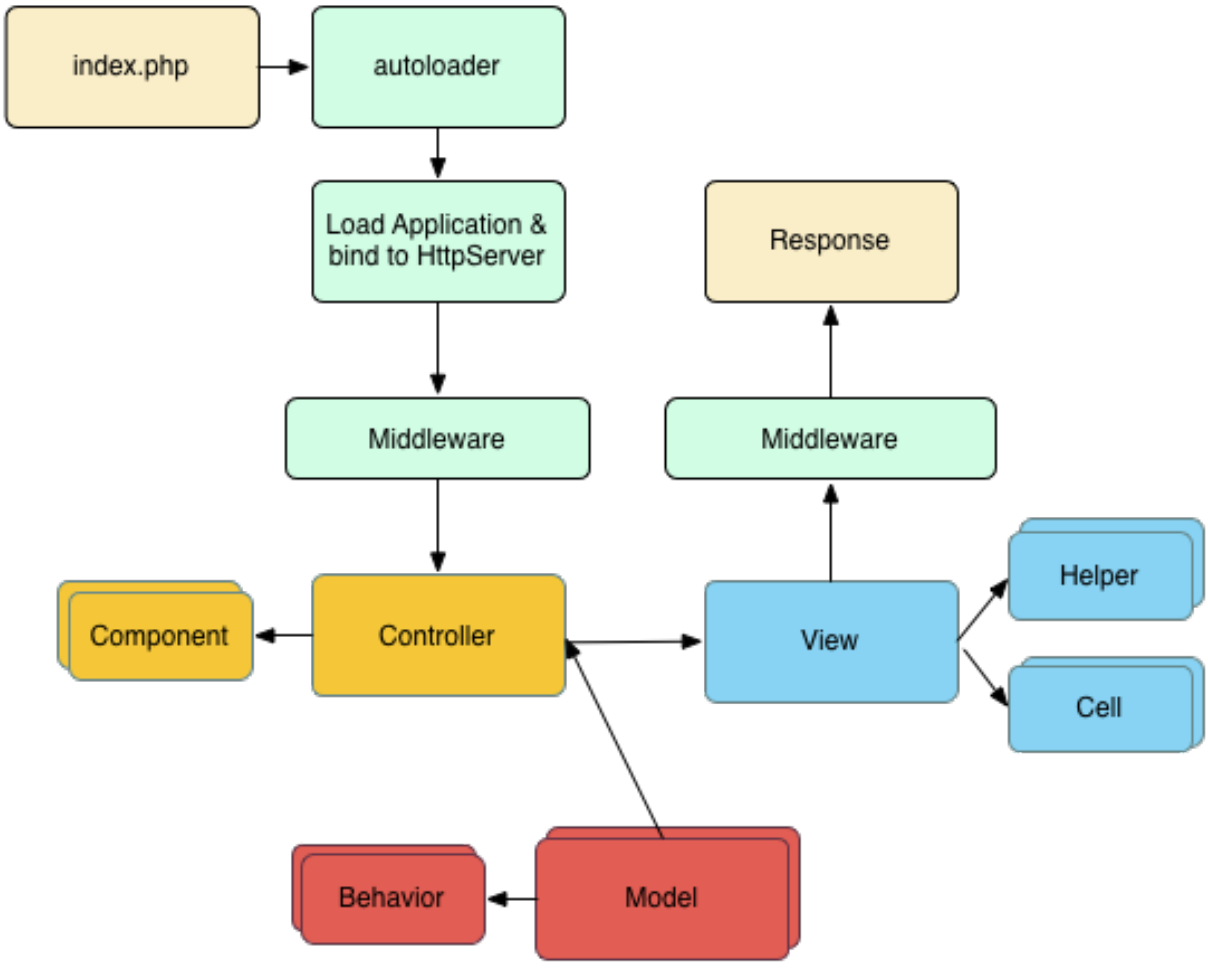
Você pode perceber que nós nunca renderizamos uma view explicitamente. As convenções do CakePHP tomarão o cuidado de selecionar a view correta e renderizá-la com os dados definidos com `set()`.

Ciclo de Requisições do CakePHP

Agora que você está familiarizado com as diferentes camadas no CakePHP, veja na imagem abaixo como o ciclo de requisição funciona:

O ciclo de requisição do CakePHP começa com a solicitação de uma página ou recurso da sua aplicação, seguindo a sequência 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* solicitados.
7. O *controller* delega a responsabilidade de criar respostas à *view* para gerar a saída de dados resultante do *Model*.
8. A *View* utiliza *Helpers* e *Cells* para gerar o corpo e o cabeçalho da resposta.
9. A resposta é enviada de volta através do *Middleware*.
10. O `HttpServer` envia a resposta para o servidor web.


index.php
autoloader
Load Application & bind to HttpServer
Middleware
Controller
Component
Behavior
Model
View
Middleware
Response
Helper
Cell

Apenas o Começo

Esperamos que essa rápida visão geral tenha despertado seu interesse. Alguns outros grandes recursos do CakePHP são:

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

Os próximos passos óbvios são *baixar o CakePHP* e 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, especiamente nos horários diurnos da América do Sul e América do Norte. Nós apreciaríamos ouví-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.

Onde conseguir ajuda em sua língua

Francês

- Comunidade CakePHP francesa[6]

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.

[4] https://cakephp.org/team
[5] https://stackoverflow.com/questions/tagged/cakephp/
[6] https://cakephp-fr.org

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:

```
$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 *Conectando Rotas*.

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 **EspeciallyFunkableBehavior** deveria ser encontrada em um arquivo nomeado **EspeciallyFunkableBehavior.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`, respectivamente.

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 ou hasOne 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 **templates/People/get_ready.php**. O padrão é **templates/Controller/underscored_function_name.php**.

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 **templates/People/index.php**

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 disponbilizado (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 tivé-lo feito.
- O diretório *webroot* será a raíz pública de documentos da sua aplicação. Ele contem 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.

CAPÍTULO 2

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 subir nossa aplicação de forma rápida.

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[7] oficial.

[7] https://getcomposer.org/download/

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:4.* 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 checadas 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 em quase todos os lugares, tornando mais fácil construir aplicações multi-arrendados.

Os nomes de tabelas e colunas que usamos não foram arbitrárias. 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 e 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[8] 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'],
```

(continues on next page)

[8] https://codahale.com/how-to-safely-store-a-password/

(continuação da página anterior)

```
    function ($routes) {
        $routes->connect('/tagged/*', ['action' => 'tags']);
    }
);
```

O trecho 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 trecho:

```
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:

```
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 **templates/Bookmarks/tags.php** 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 todos os usuários. 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 tags que possuam.

Adicionando login

No CakePHP, a autenticação é feita por *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 se ligar ao evento do ciclo de vida do controller e interagir com a sua aplicação. Para começar, vamos adicionar o AuthComponent a nossa aplicação. É essencial que cada método exija autenticação, por isso vamos acrescentar o *AuthComponent* em nosso AppController:

```
// 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 você 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
```

(continues on next page)

(continuação da página anterior)

```
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 **templates/Users/login.php** adicione o seguinte trecho:

```
<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 formulário de login simples, 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.');

    return $this->redirect($this->Auth->logout());
}
```

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 trecho:

```
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 se conectar, 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 `AppController` adicione o seguinte:

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

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

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

Seu método `initialize` agora deve parecer com:

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

(continues on next page)

(continuação da página anterior)

```
    // 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 templates/layout/default.php
// 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.

Primeiramente, vamos refatorar o formulário de adição. Para começar remova o `input('user_id')` a partir de **templates/Bookmarks/add.php**. 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.');

            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'));
}
```

Ao definir a propriedade da entidade com os dados da sessão, nós removemos qualquer possibilidade do user modificar algo que não pertenca a ele. 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'),
        ]
    ];
```

(continues on next page)

(continuação da página anterior)

```
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
}
```

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

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->_fields['tag_string'])) {
        return $this->_fields['tag_string'];
    }
    if (empty($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 array $_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 *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.

CAPÍTULO 3

4.0 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[9] 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.

[9] https://github.com/cakephp/docs

CAPÍTULO 4

Tutoriais & Exemplos

Nesta seção, você poderá caminhar através de típicas aplicações CakePHP para ver como todas as peças se encaixam.

Alternativamente, você pode seguir direto para o repositório não oficial de plugins para o CakePHP CakePackages[10] e a Bakery (Padaria)[11] para conhecer aplicações e componentes existentes.

Tutorial - Gerenciador de Conteúdo

Este tutorial irá orientá-lo através da criação de uma simples aplicação do tipo CMS (Sistema Gerenciador de Conteúdo). Para começar, nós iremos instalar o CakePHP, criar nosso banco de dados e construir um gerenciador de artigos.

Você vai precisar:

1. Um servidor de banco de dados. Nós utilizaremos um servidor MySQL neste tutorial. Você precisará saber o suficiente de SQL para criar o banco de dados e executar trechos de SQL deste tutorial. O CakePHP cuidará de criar todas as queries que sua aplicação precisará. Considerando que estamos usando MySQL, confirme que você possui a extensão `pdo_mysql` habilitada no PHP.
2. Conhecimento básico de PHP.

Antes de começar, verifique se você está usando uma versão atualizada do PHP:

```
php -v
```

Sua versão do PHP precisa ser no mínimo 8.1 (CLI) ou superior. A versão PHP do seu servidor web ta'bme precisa ser no mínimo 8.1 ou superior, e deve ser a mesma versão encontrada no terminal de comando (CLI).

[10] https://plugins.cakephp.org/
[11] https://bakery.cakephp.org/

Instalando 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, caso você já não o tenha. Se possuir instalado o programa *cURL*, basta executar o seguinte comando:

```
.. code-block:: console
```

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

Você também pode baixar o arquivo `composer.phar` do site[12] 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 **cms**.

```
.. code-block:: console
```

php composer.phar create-project –prefer-dist cakephp/app:5.* cms

Caso você tenha feito o download e executado o Instalador para Windows do Composer[13], então digite a linha abaixo no seu terminal de dentro do diretório de instalação (ex. C:\wamp\www\dev):

```
composer self-update && composer create-project --prefer-dist cakephp/app:4.* cms
```

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:

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

[12] https://getcomposer.org/download/
[13] https://getcomposer.org/Composer-Setup.exe

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*.

Caso tenha dificuldades durante este tutorial, você pode ver o resultado final no GitHub[14].

Verificando sua Instalação

Podemos verificar rapidamente se nossa instalação está correta acessando a página inicial padrão. Mas antes de poder acessar, você precisa iniciar um servidor de desenvolvimento:

```
cd /path/to/our/app

bin/cake server
```

Nota: No Windows, o comando precisa ser `bin\cake server` (repare as barras invertidas).

Isso iniciará o servidor web embutido do PHP na porta 8765. Abra **http://localhost:8765** no seu navegador web para ver a página de boas vindas. Todos os tópicos devem ter chapéus de chef verdes, exceto diz sobre o CakePHP estar apto a acessar seu banco de dados. Caso contrário, voc%e pode precisar instalar alguma extensão PHP ou definir permissão de diretórios.

A seguir, nós iremos construir o *Banco de Dados e criar nosso primeiro modelo*.

Tutorial - Gerenciador de Conteúdo - Criando o Banco de Dados

Agora que temos o CakePHP instalado, vamos configurar o banco de dados para nossa aplicação CMS. Se você ainda não tiver feito, crie um banco de dados vazio para usar neste tutorial, com um nome de sua escolha, ex. `cake_cms`. Se você está usando MySQL/MariaDB, você pode executar o SQL abaixo para criar as tabelas necessárias:

```
USE cake_cms;

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 articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (slug),
```

(continues on next page)

[14] https://github.com/cakephp/cms-tutorial

(continuação da página anterior)

```
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
) CHARSET=utf8mb4;

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (title)
) CHARSET=utf8mb4;

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY article_key(article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', 1, NOW(), NOW());
```

Se você está usando PostgreSQL, conecte ao banco de dados `cake_cms` e execute o SQL abaixo:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created TIMESTAMP,
    modified TIMESTAMP
);

CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (slug),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
```

(continues on next page)

(continuação da página anterior)

```
    title VARCHAR(191),
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (title)
);

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY (tag_id) REFERENCES tags(id),
    FOREIGN KEY (article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', TRUE, NOW(), NOW());
```

Você deve ter reparado que a tabela `articles_tags` usa uma chave primária composta. CakePHP suporta chaves primária compostas praticamente em todo lugar permitindo que você tenha esquemas que não dependem de uma coluna adicional `id`.

O nome das tabelas e colunas que nós utilizamos não são arbitrárias. Ao utilizar a *convenção de nomes*, nós aproveitamos melhor o CakePHP e evitamos a necessidade de configurar o framewok. Apesar do CakePHP ser flexível suficiente para atender praticamente todo esquema de banco de dados, aderindo as convenções você economizará seu tempo aproveitando a convenção baseada em valores padrões que o CakePHP oferece.

Nota: Os nomes das tabelas e colunas estão em inglês para que as convenções funcionem sem nenhuma configuração extra, mas é possível *configurar as inflexões* do CakePHP para reconhecer as convenções em português.

Configuração do Banco de Dados

A seguir vamos dizer ao CakePHP onde nosso banco de dados está e como se conectar a ele. Substitua os valores no array `Datasources.default` dentro do arquivo **config/app_local.php** pelos que se aplicam a sua instalação. Um exemplo completo de como deve ficar o array de configuração segue abaixo:

```
<?php
return [
    // Mais configurações acima.
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            // Substitua Mysql por Postgres se você estiver usando PostgreSQL
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
```

(continues on next page)

(continuação da página anterior)

```
            'username' => 'cakephp',
            'password' => 'sua_senha',
            'database' => 'cake_cms',
            // Comente a linha abaixo se estiver usando PostgreSQL
            'encoding' => 'utf8mb4',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
        ],
    ],
    // Mais configurações abaixo.
];
```

Uma vez que você tenha salvo seu arquivo **config/app_local.php**, você deve ver a mensagem "CakePHP is able to connect to the database" com o chapéu de chefe na cor verde.

Nota: Se você não tiver o arquivo **config/app_local.php** na sua aplicação, você deve configurar sua conexão no arquivo **config/app.php**.

Criando nosso Primeiro Modelo

Modelos são o coração de uma aplicação CakePHP. Ele permite a nós ler e escrever nossos dados. Eles possibilitam a criação de relacionamentos entre nossos dados, validar dados, e aplicar regras da aplicação. Modelos formam a fundação necessária para construir nossas ações de controles e templates.

Modelos no CakePHP são compostos dos objetos `Table` (Tabela) e `Entity` (Entidade). Objetos `Table` fornecem acesso a coleção de entidades armazenadas em uma tabela específica. Elas ficam salvas em **src/Model/Table**. O arquivo que iremos criar ficará salvo em **src/Model/Table/ArticlesTable.php**. O arquivo completo deve se parecer com isso:

```
<?php
// src/Model/Table/ArticlesTable.php
namespace App\Model\Table;

use Cake\ORM\Table;

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

Nós vinculamos o behavior *Timestamp* que irá preencher automaticamente as colunas `created` (criado) e `modified` (modificado) de nossa tabela. Ao nomear nosso objeto Table `ArticlesTable`, o CakePHP se baseia nas convenções de nomes para saber que nosso modelo utiliza a tabela `articles`. O CakePHP também usa as convenções para saber que a coluna `id` é a chave primária da tabela.

Nota: O CakePHP criará dinamicamente um objeto de modelo para você se ele não conseguir encontrar o arquivo correspondente em **src/Model/Table**. Isso significa que se você acidentalmente nomear errado o arquivo (ex. arti-

clestable.php ou ArticleTable.php), o CakePHP não reconhecerá nenhuma de suas configurações e utilizará o modelo dinâmicamente gerado no lugar.

Nós também vamos criar uma classe Entity para nossa Articles. Entidades representam um único registro do nosso banco de dados, e implementam comportamento a nível de linha para nossos dados. Nossa entidade será salva em **src/Model/Entity/Article.php**. O arquivo complete deve parecer com este:

```
<?php
// src/Model/Entity/Article.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        '*' => true,
        'id' => false,
        'slug' => false,
    ];
}
```

Nossa entidade está bem curta agora, e nós iremos configurar apenas a propriedade `_accessible` que controla quais propriedades podem ser modificadas com *Atribuição em Massa*.

Nós não podemos fazer muito com nossos modelos agora, então a seguir iremos criar nossos *Controller e Template* que nos permitirá interagir com nosso modelo.

CMS Tutorial - Creating the Articles Controller

TODO

CMS Tutorial - Authentication

TODO

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 subir nossa aplicação de forma rápida.

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[15] 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:4.* 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*.

[15] https://getcomposer.org/download/

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 checadas 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 em quase todos os lugares, tornando mais fácil construir aplicações multi-arrendados.

Os nomes de tabelas e colunas que usamos não foram arbitrárias. 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 e 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[16] 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:

[16] https://codahale.com/how-to-safely-store-a-password/

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

O trecho 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 trecho:

```
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:

```
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 **templates/Bookmarks/tags.php** 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 todos os usuários. 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 tags que possuam.

Adicionando login

No CakePHP, a autenticação é feita por *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 se ligar ao evento do ciclo de vida do controller e interagir com a sua aplicação. Para começar, vamos adicionar o AuthComponent a nossa aplicação. É essencial que cada método exija autenticação, por isso vamos acrescentar o *AuthComponent* em nosso AppController:

```
// 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 você 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
```

(continues on next page)

(continuação da página anterior)

```
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 **templates/Users/login.php** adicione o seguinte trecho:

```
<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 formulário de login simples, 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.');

    return $this->redirect($this->Auth->logout());
}
```

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 trecho:

```
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 se conectar, 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 `AppController` adicione o seguinte:

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

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

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

Seu método `initialize` agora deve parecer com:

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

(continues on next page)

(continuação da página anterior)

```
    // 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 templates/layout/default.php
// 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.

Primeiramente, vamos refatorar o formulário de adição. Para começar remova o `input('user_id')` a partir de **templates/Bookmarks/add.php**. 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.');

            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'));
}
```

Ao definir a propriedade da entidade com os dados da sessão, nós removemos qualquer possibilidade do user modificar algo que não pertenca a ele. 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'),
        ]
    ];
```

(continues on next page)

(continuação da página anterior)

```
    $this->set('bookmarks', $this->paginate($this->Bookmarks));
}
```

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

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->_fields['tag_string'])) {
        return $this->_fields['tag_string'];
    }
    if (empty($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 array $_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 *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 8.1 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, caso você já não o tenha. Se possuir instalado o programa *cURL*, basta executar o seguinte comando:

```
.. code-block:: console
```

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

Você também pode baixar o arquivo `composer.phar` do site[17] 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:4.* [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:

[17] https://getcomposer.org/download/

```
/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*.

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 -R 777 tmp
chmod -R 777 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,
```

(continues on next page)

(continuação da página anterior)

```
    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 arbitrárias. 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, artciclestable.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 /bake/usage 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 /bake/usage 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, localizado em **templates/layout/default.php**.

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: templates/Articles/index.php -->

<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>
```

(continues on next page)

(continuação da página anterior)

```
        <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 especifíca 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')` porquê nós queremos a informação de apenas um artigo.

Repare que nossa action recebe um parâmetro: o ID do artigo que gostariamos 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 **templates/Articles/view.php**

```
<!-- File: templates/Articles/view.php -->

<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 = $this->Articles->newEmptyEntity();
        if ($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.'));
```

(continues on next page)

(continuação da página anterior)

```
        }
        $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[18] 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 API[19] 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.

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: templates/Articles/add.php -->

<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();
?>
```

[18] https://php.net/manual/en/language.oop5.overloading.php#object.call
[19] https://api.cakephp.org

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 **templates/Articles/index.php** 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: templates/Articles/edit.php -->

<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: templates/Articles/index.php  (edit links added) -->

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

(continues on next page)

(continuação da página anterior)

```
        <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: templates/Articles/index.php (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 `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 *Conectando Rotas*.

Por padrão, o CakePHP responde a uma requisição pela raíz do seu site usando o `PagesController`, ao renderizar uma view chamada **home.php**. 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. /bake/usage: Gerando código CRUD
4. ***Tutorial - Criando um Blog - Autenticação e Autorização*: Tutorial de** autorização e autenticação

Tutorial - Criando um Blog - Parte 3

Criar uma arvore 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, addicione atráves 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,
```

(continues on next page)

(continuação da página anterior)

```
            '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]␣
↪rght: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('rght', '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,
        ]);
```

(continues on next page)

(continuação da página anterior)

```
        $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 criadas, 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 **templates/Categories/add.php** e **templates/Categories/edit.php**:

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

Anexar árvore de compartamento 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 colunas rght:

```
$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('rght');
```

Além disso, você deve desabilitar ou remover o requirePresence do validador, tanto para a `lft` e `rght` 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')
```

(continues on next page)

(continuação da página anterior)

```
        ->notEmpty('lft');

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

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 TreeBahavior

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.
```

(continues on next page)

(continuação da página anterior)

```
↪');
        }

        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

Em templates/Categories/index.php 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>
```

(continues on next page)

(continuação da página anterior)

```
    <?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;

// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\Exception\NotFoundException;

class ArticlesController extends AppController
{
    // ...

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($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: templates/Articles/add.php -->

<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,
    email VARCHAR(255),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Respeitado as convenções do CakePHP para nomear tabelas, mas também aproveitando de outras convenção: Usando as colunas `email` 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
{
```

(continues on next page)

(continuação da página anterior)

```
    public function validationDefault(Validator $validator)
    {
        return $validator
            ->notEmpty('email', 'Email é necessário')
            ->email('email')
            ->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 atráves 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)) {
```

(continues on next page)

(continuação da página anterior)

```
                $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.php:

```
<!-- templates/Users/add.php -->

<div class="users form">
<?= $this->Form->create($user) ?>
    <fieldset>
        <legend><?= __('Add User') ?></legend>
        <?= $this->Form->input('email') ?>
        <?= $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 arquivos **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()
    {
```

(continues on next page)

(continuação da página anterior)

```
        $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`, estás 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 `email` 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 `AuthComponet` 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());
        }
```

(continues on next page)

(continuação da página anterior)

```
        $this->Flash->error(__('Usuário ou senha ínvalido, 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 array $_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 **templates/Users/login.php** e adicione as seguintes linhas:

```
<!-- File: templates/Users/login.php -->

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

(continues on next page)

(continuação da página anterior)

```
<?= $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 `email` 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 = $this->Articles->newEmptyEntity();
    if ($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 metódo 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 está 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 metódo `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;
```

(continues on next page)

(continuação da página anterior)

```
    }

    // Apenas o proprietário do artigo pode editar e excluí
    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 *AuthComponent* seção onde você encontrará mais sobre a configuração do componente, criação de classes de Autorização customizadas, e muito mais.

Sugerimos as seguintes leituras

1. /bake/usage Generating basic CRUD code
2. *AuthComponent*: User registration and login

CAPÍTULO 5

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/traduções 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[20], sendo assim, você pode checar o status de várias builds[21] 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.

[20] https://en.wikipedia.org/wiki/Continuous_integration
[21] https://ci.cakephp.org

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)
- /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, sinta-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.
- Sinta-se a vontade para mergulhar na tradução caso o seu idioma já exista no manual.
- Use Linguagem Informal[22].
- 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* `<em>`. 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.
- Não use Entidades HTML[23] 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 paral tal antes de incluí-la em seu idioma.

Guia de formatação para documentação

A nova documentação do CakePHP é escrita com ReST[24]. 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.

[22] https://pt.wikipedia.org/wiki/Linguagem_coloquial
[23] https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references
[24] https://en.wikipedia.org/wiki/ReStructuredText

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 asteríscos: **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 «()».
  - ``` ``cascadeCallbacks`` ```, ``` ``true`` ```, ``` ``id`` ```, ``` ``PagesController`` ```, ``` ``config()`` ```, etc.

Se asteríscos ou *backquotes* aparecerem em texto corrido e ficarem confusos com delimitadores de maração 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 asterísco 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 diveros tipos de *links*, cada um com usos particulares.

Links externos

Links para documentos externos podem ser feitos desta forma:

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

O link resultante ficaria assim: Link externo para php.net[25]

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* referenciaria `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>``.

[25] https://php.net

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[26] 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.

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
```

[26] https://pypi.org/project/sphinxcontrib-phpdomain/

Veja 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 4.0.0: Esse recurso incrível foi adicionado na versão 4.0.0

Obsoleto desde a versão 4.0.1: Esse recurso antigo foi descontinuado na versão 4.0.1

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[27].

Reportando bugs

Relatórios de *bugs* bem escritos são muito úteis. Existem algumas medidas que ajudam a criar relatórios de erro melhores:

- **Faça**: Busque[28] 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[29] possui muitos desenvolvedores dispníveis para ajudar a responder suas dúvidas. Também dê uma olhada no Stack Overflow[30].

[27] https://github.com/cakephp/cakephp/issues
[28] https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues
[29] https://webchat.freenode.net
[30] https://stackoverflow.com/questions/tagged/cakephp

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*.

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 8.1 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[31].

Clone o código-fonte do CakePHP do Github:

- Se você não tem uma conta no GitHub[32], crie uma.
- Dê *Fork* no repositório do CakePHP[33] clicando no botão **Fork&**.

Depois que seu *fork* for feito, clone seu *fork* para sua máquina:

```
git clone git@github.com:SEUNOME/cakephp.git
```

[31] https://git-scm.com/book/
[32] https://github.com
[33] https://github.com/cakephp/cakephp

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
```

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 prcocemente, 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 licensiado sob a licença MIT, e a Cake Software Foundation[34] será a proprietária de qualquer código proveniente de contribuição. Os contribuidores devem seguir as regras comunitárias do CakePHP[35].

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-12[36] 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[37] para verificar se o seu código segue os padrões estabelecidos.

[34] https://cakefoundation.org/old
[35] https://cakephp.org/get-involved
[36] https://www.php-fig.org/psr/psr-12/
[37] https://github.com/cakephp/cakephp-codesniffer

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.
- O parêntese de fechamento de uma função multi-linha deve ter uma linha reservada para sí.

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 melhorar 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) {
```

(continues on next page)

(continuação da página anterior)

```
    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 .php) 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()) {
    // ...
}
```

(continues on next page)

(continuação da página anterior)

```
// 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-12[38], 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 distribuidos em linhas separadas e indentados com quatro espaços:

```
$email->from('foo@exemplo.com')
    ->to('bar@exemplo.com')
    ->subject('Uma mensagem legal')
    ->send();
```

[38] https://www.php-fig.org/psr/psr-12/

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 phpDocumentor[39]:

- @deprecated[40] Usando o formato `@version <vector> <description>`, onde `version` e `description` são obrigatórios.
- @example[41]
- @ignore[42]
- @internal[43]
- @link[44]
- @see[45]
- @since[46]
- @version[47]

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
 */
function bar()
{
}

/**
 * Função foo.
 *
 * @return void
 */
function foo()
```

(continues on next page)

[39] https://phpdoc.org
[40] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/deprecated.html
[41] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/example.html
[42] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/ignore.html
[43] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/internal.html
[44] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/link.html
[45] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/see.html
[46] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/since.html
[47] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/version.html

(continuação da página anterior)

```
{
}
```

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()
```

(continues on next page)

(continuação da página anterior)

```
{
    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[48].

Tags do PHP

Use sempre *tags* longas (`<?php ?>`) ao invés de *tags* curtas (`<? ?>`). O *short echo* deve ser usado em arquivos de template (**.php**) 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()
{
}
```

[48] https://php.net/require_once

Classes

Escreva todas as funções no padrão «CamelCase», isto é, com a primeira letra de cada palavra que compõem o nome da classe maiúscula:

```
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**[49]), sendo recomendado o seu uso em documentações como exemplos.

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);
```

[49] https://datatracker.ietf.org/doc/html/rfc2606.html

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'])) {
```

(continues on next page)

(continuação da página anterior)

```
        // ...
    }

    // 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[50], 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.

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 espectativas 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.

[50] https://semver.org/

Classes

Classes oferecidas pelo CakePHP podem ser construidas 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]
Adicinar um valor padrão a um argumento de método existente	Sim

[1] Seu código *pode* ser quebrado por *minor releases*. Verifique o guia de migração para mais detalhes.

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 *minor release* 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[Página 104, 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

[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 significante.

[3] Evite sempre que possível. Qualquer remoção precisa ser documentada no guia de migração.

CAPÍTULO 6

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 8.1 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 inclusos internamente requerem PDO. Você deve assegurar-se que possui a extensão PDO correta instalada.

Instalando o CakePHP

O CakePHP utiliza Composer[51], 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[52].

Para sistemas Windows, você pode baixar o instalador aqui[53]. Mais instruções para o instalador Windows do Composer podem ser encontradas dentro do LEIA-ME aqui[54].

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:4.* [app_name]
```

Ou se o Composer estiver instalado globalmente:

```
composer create-project --prefer-dist cakephp/app:4.* [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.

[51] https://getcomposer.org
[52] https://getcomposer.org/download/
[53] https://github.com/composer/windows-setup/releases/
[54] 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 https://cwiki.apache.org/confluence/display/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 raíz de sua aplicação através do bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
```

(continues on next page)

(continuação da página anterior)

```
    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[55] 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 inváidos 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.

[55] https://clickontyler.com/virtualhostx/

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:

```
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[56] para instalar o Rewrite Module 2.0[57] ou baixe-o diretamente (32-

[56] https://www.microsoft.com/web/downloads/platform.aspx
[57] https://www.iis.net/downloads/microsoft/url-rewrite

bit[58] / 64-bit[59]).

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/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets(img, css, files, js,␣
↪favicon)"
                  stopProcessing="true">
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <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 natívas 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
```

[58] https://download.microsoft.com/download/D/8/1/D81E5DD6-1ABB-46B0-9B4B-21894E18B77F/rewrite_x86_en-US.msi
[59] https://download.microsoft.com/download/1/2/8/128E2E22-C1B9-44A4-BE2A-5859ED1D4592/rewrite_amd64_en-US.msi

Isso fará suas URLs parecem como www.example.com/index.php/controllername/actionname/param ao invés de www.example.com/controllername/actionname/param.

CAPÍTULO 7

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 seqüência 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 seqüência aleatória usada em hash. Uma seqüência 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 *Configuração da Sessão* 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
            ]
        ]
```

(continues on next page)

(continuação da página anterior)

```
    ]
];
```

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'];
```

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 escreveendo 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 PHP[60] 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:

```
Configure::drop('default');
```

[60] https://php.net/parse_ini_file

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 *Teste de Integração do Controlador* 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[61] 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/persistent/`)
- `CACHE_CAKEMODEL_URL` (ie `File:///?prefix=myapp_cake_model_&serialize=true&timeout=3600&path=./tmp/cache/models/`)
- `EMAIL_TRANSPORT_DEFAULT_URL` (ie `smtp://user:password@hostname:port?tls=null&client=null&timeout=30`)
- `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=./logs/`)
- `LOG_ERROR_URL` (ie `file:///?levels[]=warning&levels[]=error&levels[]=critical&levels[]=alert&levels[]=./logs/`)

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[62] para usar variáveis de ambiente em um desenvolvimento local. Consulte as instruções Leiame 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 implícitamente usa um `Cake\ORM\Table` genérico em vez de sua classe concreta assim:

[61] https://12factor.net/
[62] https://github.com/josegonzalez/php-dotenv

```
// No seu bootstrap.php
use Cake\Event\EventManager;
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\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);
        }
    });
}
```

CAPÍTULO 8

Roteamento

class Cake\Routing**Router**

O roteamento fornece ferramentas que mapeiam URLs para ações do controlador. Ao definir rotas, você pode separar como o aplicativo é implementado e como os URLs são estruturadas.

O roteamento no CakePHP também abrange a idéia de roteamento reverso, onde uma matriz de parâmetros pode ser transformada em uma string de URL. Ao usar o roteamento reverso, você pode redimensionar a estrutura de URL do seu aplicativo sem precisar atualizar todo o seu código.

Tour Rápido

Esta seção ensinará a você, como exemplo, os usos mais comuns do CakePHP Router. Normalmente, você deseja exibir algo como uma página de destino e adicionar isso ao seu arquivo **routes.php**:

```
use Cake\Routing\Router;

// Usando o construtor de rota com escopo.
Router::scope('/', function ($routes) {
    $routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
});

// Usando o método estático.
Router::connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

O `Router` fornece duas interfaces para conectar rotas. O método estático é uma interface compatível com versões anteriores, enquanto os construtores com escopo oferecem uma sintaxe mais concisa ao criar várias rotas e melhor desempenho.

Isso executará o método de índice no `ArticlesController` quando a página inicial do seu site for visitada. Às vezes, você precisa de rotas dinâmicas que aceitem vários parâmetros; esse seria o caso, por exemplo, de uma rota para

visualizar o conteúdo de um artigo:

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

A rota acima aceitará qualquer URL semelhante a `/articles/15` e invocará o método `view(15)` no `ArticlesController`. Porém, isso não impedirá que as pessoas tentem acessar URLs semelhantes a `/articles/foobar`. Se desejar, você pode restringir alguns parâmetros para estar em conformidade com uma expressão regular:

```
$routes->connect(
    '/articles/:id',
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// Antes de 3.5, use o array de opções
$routes->connect(
    '/articles/:id',
    ['controller' => 'Articles', 'action' => 'view'],
    ['id' => '\d+', 'pass' => ['id']]
)
```

O exemplo anterior alterou o marcador de estrelas por um novo espaço reservado para `:id`. O uso de espaços reservados nos permite validar partes da URL; nesse caso, usamos a expressão regular `\d+` para que apenas os dígitos correspondam. Finalmente, pedimos ao roteador para tratar o espaço reservado `id` como um argumento de função para o método `view()` especificando a opção `pass`. Mais sobre o uso dessa opção posteriormente.

O roteador do CakePHP também pode reverter as rotas de correspondência. Isso significa que, a partir de uma matriz que contém parâmetros correspondentes, é capaz de gerar uma string de URL:

```
use Cake\Routing\Router;

echo Router::url(['controller' => 'Articles', 'action' => 'view', 'id' => 15]);
// Saída
/articles/15
```

As rotas também podem ser rotuladas com um nome exclusivo, isso permite que você as referencie rapidamente ao criar links, em vez de especificar cada um dos parâmetros de roteamento:

```
// Em routes.php
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'login']);
// Saída
/login
```

Para ajudar a manter seu código de roteamento DRY, o roteador tem o conceito de "escopos". Um escopo define um segmento de caminho comum e, opcionalmente, rotea os padrões. Todas as rotas conectadas dentro de um escopo herdarão o caminho/padrão de seus escopos de encapsulamento:

```
Router::scope('/blog', ['plugin' => 'Blog'], function ($routes) {
    $routes->connect('/', ['controller' => 'Articles']);
});
```

A rota acima combinaria com `/blog/` e enviaria para `Blog\Controller\ArticlesController::index()`.

O esqueleto do aplicativo vem com algumas rotas para você começar. Depois de adicionar suas próprias rotas, você poderá remover as rotas padrão se não precisar delas.

Conectando Rotas

`Cake\Routing\Router::connect(`*$route*, *$defaults = []*, *$options = []*`)`

Para manter seu código *DRY* você deve usar "escopos de roteamento". Os escopos de roteamento não apenas permitem que você mantenha seu código DRY, eles também ajudam o Router a otimizar sua operação. O método padrão é o escopo `/`. Para criar um escopo e conectar algumas rotas, usaremos o método `scope()`:

```
// Em config/routes.php
use Cake\Routing\Route\DashedRoute;

Router::scope('/', function ($routes) {
    // Conecte as rotas de fallback genéricas.
    $routes->fallbacks(DashedRoute::class);
});
```

O método `connect()` leva até três parâmetros: o modelo de URL que você deseja corresponder, os valores padrão para seus elementos de rota e as opções para a rota. As opções freqüentemente incluem regras de expressões regulares para ajudar o roteador a combinar elementos na URL.

O formato básico para uma definição de rota é:

```
$routes->connect(
    '/url/template',
    ['default' => 'defaultValue'],
    ['option' => 'matchingRegex']
);
```

O primeiro parâmetro é usado para informar ao roteador que tipo de URL você está tentando controlar. A URL é uma string delimitada por uma barra normal, mas também pode conter um curinga (*) ou *Elementos de Rota*. O uso de um curinga informa ao roteador que você deseja aceitar quaisquer argumentos adicionais fornecidos. As rotas sem um * correspondem apenas ao padrão de modelo exato fornecido.

Depois de especificar uma URL, use os dois últimos parâmetros de `connect()` para dizer ao CakePHP o que fazer com uma solicitação, uma vez que ela corresponda. O segundo parâmetro é uma matriz associativa. As chaves da matriz devem ser nomeadas após os elementos de rota que o modelo de URL representa. Os valores na matriz são os valores padrão para essas chaves. Vejamos alguns exemplos básicos antes de começarmos a usar o terceiro parâmetro de `connect()`:

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

Esta rota é encontrada no arquivo routes.php distribuído com o CakePHP. Ele corresponde a qualquer URL que comece com `/pages/` e passa para a ação `display()` do `PagesController`. Um pedido para `/pages/products` seria mapeado para `PagesController->display('products')`.

Além da estrela gananciosa `/*` existe também a sintaxe da estrela `/**`. Usando uma estrela dupla à direita, capturaremos o restante de uma URL como um único argumento transmitido. Isto é útil quando você quer usar um argumento que inclua um `/` nele:

```
$routes->connect(
    '/pages/**',
    ['controller' => 'Pages', 'action' => 'show']
);
```

A URL de entrada de `/pages/the-example-/-e-proof` resultaria em um único argumento passado de `the-example-/-e-proof`.

Você pode usar o segundo parâmetro de `connect()` para fornecer quaisquer parâmetros de roteamento que sejam compostos dos valores padrão da rota

```
$routes->connect(
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

Este exemplo mostra como você pode usar o segundo parâmetro de `connect()` para definir parâmetros padrão. Se você criou um site que apresenta produtos para diferentes categorias de clientes, considere a possibilidade de criar uma rota. Isso permite que você crie um link para `/government` em vez de `/pages/display/5`.

Um uso comum para o roteamento é criar segmentos de URL que não correspondam aos seus nomes de controlador ou modelo. Digamos que em vez de acessar nosso URL regular em */users/some_action/5* `, gostaríamos de poder acessá-lo por `/cooks/some_action/5`. A rota seguinte cuida disso:

```
$routes->connect(
    '/cooks/:action/*', ['controller' => 'Users']
);
```

Isto está dizendo ao Roteador que qualquer URL que comece com `/cooks/` deve ser enviado para o `UsersController`. A ação chamada dependerá do valor do parâmetro `:action`. Usando *Elementos de Rota*, você pode criar rotas variáveis, que aceitam entrada ou variáveis do usuário. A rota acima também usa a estrela gananciosa. A estrela gananciosa indica que esta rota deve aceitar qualquer argumento de posição adicional dado. Estes argumentos serão disponibilizados no array *Passando Argumentos*.

Ao gerar URLs, as rotas são usadas também. Usando `['controller' => 'Users', 'action' => 'some_action', 5]` como uma URL irá gerar `/cooks/some_action/5` se a rota acima for a primeira encontrada.

As rotas que conectamos até agora corresponderão a qualquer verbo HTTP. Se você estiver criando uma API REST, geralmente desejará mapear ações HTTP para diferentes métodos de controlador. O `RouteBuilder` fornece métodos auxiliares que tornam mais simples a definição de rotas para verbos HTTP específicos:

```
// Crie uma rota que responda apenas a solicitações GET.
$routes->get(
    '/cooks/:id',
    ['controller' => 'Users', 'action' => 'view'],
    'users:view'
);
```

(continues on next page)

(continuação da página anterior)

```
// Criar uma rota que responda apenas a solicitações PUT
$routes->put(
    '/cooks/:id',
    ['controller' => 'Users', 'action' => 'update'],
    'users:update'
);
```

As rotas acima mapeiam a mesma URL para diferentes ações do controlador com base no verbo HTTP usado. As solicitações GET irão para a ação "ver", enquanto as solicitações PUT irão para a ação "atualizar". Existem métodos auxiliares HTTP para:

- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- HEAD

Todos esses métodos retornam a instância da rota, permitindo que você aproveite os *fluent setters* para configurar ainda mais sua rota.

Elementos de Rota

Você pode especificar seus próprios elementos de rota e isso permite que você defina locais na URL onde os parâmetros das ações do controlador devem estar. Quando um pedido é feito, os valores para estes elementos de rota são encontrados em `$this->request->getParam()` no controlador. Quando você define um elemento de rota personalizado, você pode, opcionalmente, especificar uma expressão regular - isso diz ao CakePHP como saber se a URL está formada corretamente ou não. Se você optar por não fornecer uma expressão regular, qualquer caractere que não seja `/` será tratado como parte do parâmetro:

```
$routes->connect(
    '/:controller/:id',
    ['action' => 'view']
)->setPatterns(['id' => '[0-9]+']);

// Antes de 3.5, use o array de opções
$routes->connect(
    '/:controller/:id',
    ['action' => 'view'],
    ['id' => '[0-9]+']
);
```

O exemplo acima ilustra como criar uma maneira rápida de visualizar modelos de qualquer controlador criando uma URL que se parece com `/controllerername/:id`. A URL fornecida para `connect()` especifica dois elementos de rota: `:controller` e `:id`. O elemento `:controller` é um elemento de rota padrão do CakePHP, portanto o roteador sabe como combinar e identificar os nomes dos controladores nas URLs. O elemento `:id` é um elemento de rota personalizado e deve ser esclarecido ainda mais especificando uma expressão regular correspondente no terceiro parâmetro de `connect()`.

O CakePHP não produz automaticamente URLs em minúsculas e tracejadas ao usar o parâmetro `:controller`. Se você precisar disso, o exemplo acima pode ser reescrito da seguinte maneira:

```
use Cake\Routing\Route\DashedRoute;

// Crie um construtor com uma classe de rota diferente.
$routes->scope('/', function ($routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/:controller/:id', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    // Antes de 3.5 usar matriz de opções
    $routes->connect(
        '/:controller/:id',
        ['action' => 'view'],
        ['id' => '[0-9]+']
    );
});
```

A classe `DashedRoute` garantirá que os parâmetros `:controller` e `:plugin` estejam corretamente em minúsculas e tracejados.

Se você precisar de URLs minúsculas e sublinhadas durante a migração de um aplicativo CakePHP 2.x, poderá usar a classe `InflectedRoute`.

Nota: Padrões usados para elementos de rota não devem conter nenhum grupo de captura. Em caso afirmativo, o roteador não funcionará corretamente.

Uma vez definida essa rota, solicitar `/apples/5` chamaria o método `view()` de ApplesController. Dentro do método `view()`, você precisaria acessar o ID passado em `$this->request->getParam('id')`.

Se você possui um único controlador no seu aplicativo e não deseja que o nome do controlador apareça na URL, é possível mapear todos os URLs para ações no seu controlador. Por exemplo, para mapear todos os URLs para ações do controlador `home`, por exemplo, ter URLs como `/demo` em vez de `/home/demo`, você pode fazer o seguinte:

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

Se você deseja fornecer um URL que não diferencia maiúsculas de minúsculas, pode usar modificadores embutidos de expressão regular:

```
// Antes da 3.5, use a matriz de opções em vez de setPatterns()
$routes->connect(
    '/:userShortcut',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

Mais um exemplo e você será um profissional de roteamento:

```
// Antes da 3.5, use a matriz de opções em vez de setPatterns()
$routes->connect(
    '/:controller/:year/:month/:day',
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
```

(continues on next page)

(continuação da página anterior)

```
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|[12][0-9]|3[01]'
]);
```

Isso está bastante envolvido, mas mostra como as rotas podem ser poderosas. O URL fornecido possui quatro elementos de rota. O primeiro é familiar para nós: é um elemento de rota padrão que diz ao CakePHP para esperar um nome de controlador.

Em seguida, especificamos alguns valores padrão. Independentemente do controlador, queremos que a ação `index()` seja chamada.

Por fim, especificamos algumas expressões regulares que corresponderão a anos, meses e dias na forma numérica. Observe que parênteses (agrupamento) não são suportados nas expressões regulares. Você ainda pode especificar alternativas, como acima, mas não agrupadas entre parênteses.

Uma vez definida, essa rota corresponderá a `/articles/2007/02/01`, `/articles/2004/11/16`, entregando as solicitações às ações `index()` de seus respectivos controladores , com os parâmetros de data em `$this->request->getParam()`.

Existem vários elementos de rota que têm um significado especial no CakePHP e não devem ser usados, a menos que você queira o significado especial

- `controller` Usado para nomear o controlador para uma rota.
- `action` Usado para nomear a ação do controlador para uma rota.
- `plugin` Usado para nomear o plug-in em que um controlador está localizado.
- `prefix` Usado para *Prefix Routing*
- `_ext` Usado para *File extentions routing*.
- `_base` Defina como `false` para remover o caminho base da URL gerada. Se o seu aplicativo não estiver no diretório raiz, isso poderá ser usado para gerar URLs que são "relativos ao cake"
- `_scheme` Configure para criar links em diferentes esquemas, como *webcal* ou *ftp*. O padrão é o esquema atual.
- `_host` Defina o host a ser usado para o link. O padrão é o host atual.
- `_port` Defina a porta se precisar criar links em portas não padrão.
- `_full` Se `true`, a constante *FULL_BASE_URL* será anexada aos URLs gerados
- `#` Permite definir fragmentos de hash de URL.
- `_https` Defina como `true` para converter o URL gerado em https ou `false` para forçar http.
- `_method` Defina o verbo/método HTTP a ser usado. Útil ao trabalhar com *Criando rotas RESTful*.
- `_name` Nome da rota. Se você configurou rotas nomeadas, poderá usar esta chave para especificá-la.

Configurando opções de rota

Há várias opções de rotas que podem ser definidas individualmente. Após conectar uma rota, você pode usar seus métodos fluentes do construtor para configurar ainda mais a rota. Esses métodos substituem muitas das chaves no parâmetro `$options` de `connect()`:

```
$routes->connect(
    '/:lang/articles/:slug',
    ['controller' => 'Articles', 'action' => 'view'],
```

(continues on next page)

(continuação da página anterior)

```
)
// Permite requisições GET e POSTS.
->setMethods(['GET', 'POST'])

// Corresponder apenas no subdomínio do blog.
->setHost('blog.example.com')

// Defina os elementos da rota que devem ser convertidos em argumentos passados
->setPass(['slug'])

// Definir os padrões correspondentes para elementos de rota
->setPatterns([
    'slug' => '[a-z0-9-_]+',
    'lang' => 'en|fr|es',
])

// Permitir também extensões de arquivo JSON
->setExtenions(['json'])

// Defina lang como um parâmetro persistente
->setPersist(['lang']);
```

Passando parâmetros para ação

Ao conectar rotas usando *Elementos de Rota*, você pode querer que elementos roteados sejam passados por argumentos. A opção `pass` lista as permissões que elementos de rota também devem ser disponibilizados como argumentos passados para as funções do controlador:

```
// src/Controller/BlogsController.php
public function view($articleId = null, $slug = null)
{
    // Algum código aqui...
}

// routes.php
Router::scope('/', function ($routes) {
    $routes->connect(
        '/blog/:id-:slug', // E.g. /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Define os elementos da rota no modelo de rota
    // para passar como argumentos de função. O pedido é importante,
    // pois isso simplesmente mapeie ":id" para $articleId em sua ação
    ->setPass(['id', 'slug'])
    // Define um padrão que o `id` deve corresponder.
    ->setPatterns([
        'id' => '[0-9]+',
    ]);
});
```

Agora, graças aos recursos de roteamento reverso, você pode passar a matriz de URLs como abaixo e o CakePHP saberá como formar a URL conforme definido nas rotas:

```
// view.php
// Isso retornará um link para /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
]);

// Você também pode usar parâmetros indexados numericamente.
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    3,
    'CakePHP_Rocks'
]);
```

Usando Rotas Nomeadas

Às vezes, você encontrará a digitação de todos os parâmetros de URL de uma rota muito detalhados ou gostaria de aproveitar as melhorias de desempenho que as rotas nomeadas possuem. Ao conectar rotas, você pode especificar uma opção `_name`, essa opção pode ser usada no roteamento reverso para identificar a rota que você deseja usar:

```
// Conecte uma rota com um nome.
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

// Nomear uma rota específica do verbo (3.5.0+)
$routes->post(
    '/logout',
    ['controller' => 'Users', 'action' => 'logout'],
    'logout'
);

// Gere um URL usando uma rota nomeada.
$url = Router::url(['_name' => 'logout']);

// Gere um URL usando uma rota nomeada,
// com algumas cadeias de caracteres de consulta args.
$url = Router::url(['_name' => 'login', 'username' => 'jimmy']);
```

Se o seu modelo de rota contiver elementos de rota como `:controller`, você precisará fornecê-los como parte das opções para `Router::url()`.

Nota: Os nomes das rotas devem ser exclusivos em todo o aplicativo. O mesmo `_name` não pode ser usado duas vezes, mesmo que os nomes ocorram dentro de um escopo de roteamento diferente.

Ao criar rotas nomeadas, você provavelmente desejará seguir algumas convenções para os nomes das rotas. O CakePHP

facilita a criação de nomes de rotas, permitindo definir prefixos de nomes em cada escopo:

```
Router::scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
    // O nome desta rota será `api:ping`
    $routes->get('/ping', ['controller' => 'Pings'], 'ping');
});

// Gere uma URL para a rota de ping
Router::url(['_name' => 'api:ping']);

// Use namePrefix com plugin()
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    // Conecta rotas.
});

// Ou com prefix()
Router::prefix('Admin', ['_namePrefix' => 'admin:'], function ($routes) {
    // Conecta rotas.
});
```

Você também pode usar a opção `_namePrefix` dentro de escopos aninhados e funciona conforme o esperado:

```
Router::plugin('Contacts', ['_namePrefix' => 'contacts:'], function ($routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function ($routes) {
        // O nome desta rota será `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings'], 'ping');
    });
});

// Gere uma URL para a rota de ping
Router::url(['_name' => 'contacts:api:ping']);
```

As rotas conectadas nos escopos nomeados somente terão nomes adicionados se a rota também for nomeada. As rotas sem nome não terão o `_namePrefix` aplicado a elas.

Prefix Routing

static Cake\Routing\Router::**prefix**(*$name*, *$callback*)

Muitos aplicativos requerem uma seção de administração na qual usuários privilegiados podem fazer alterações. Isso geralmente é feito por meio de uma URL especial, como `/admin/users/edit/5`. No CakePHP, o roteamento de prefixo pode ser ativado usando o método de escopo `prefix`:

```
use Cake\Routing\Route\DashedRoute;

Router::prefix('Admin', function ($routes) {
    // Todas as rotas aqui serão prefixadas com `/admin`. Também
    // será adicionado o elemento de rota `'prefix' => 'Admin'`,
    // que será necessário ao gerar URLs para essas rotas
    $routes->fallbacks(DashedRoute::class);
});
```

Os prefixos são mapeados para sub-namespaces no namespace `Controller` do seu aplicativo. Por ter prefixos como controladores separados, você pode criar controladores menores e mais simples. O comportamento co-

mum aos controladores prefixados e não prefixados pode ser encapsulado usando herança, *Componentes* ou traits. Usando o exemplo de nossos usuários, acessar a URL `/admin/users/edit/5` chamaria o método `edit()` do nosso **src/Controller/Admin/UsersController.php** passando 5 como o primeiro parâmetro. O arquivo de visualização usado seria **templates/Admin/Users/edit.php**.

Você pode mapear a URL /admin para sua ação `index()` do controlador de páginas usando a seguinte rota:

```
Router::prefix('Admin', function ($routes) {
    // Como você está no escopo do administrador,
    // não é necessário incluir o prefixo /admin
    // ou o elemento de rota do administrador.
    $routes->connect('/', ['controller' => 'Pages', 'action' => 'index']);
});
```

Ao criar rotas de prefixo, você pode definir parâmetros de rota adicionais usando o argumento `$options`:

```
Router::prefix('Admin', ['param' => 'value'], function ($routes) {
    // As rotas conectadas aqui são prefixadas com '/admin' e
    // têm a chave de roteamento 'param' definida.
    $routes->connect('/:controller');
});
```

Os prefixos de várias palavras são convertidos por padrão usando inflexão dasherize, ou seja, `MyPrefix` seria mapeado para `my-prefix` na URL. Certifique-se de definir um caminho para esses prefixos se você quiser usar um formato diferente, como por exemplo, sublinhado:

```
Router::prefix('MyPrefix', ['path' => '/my_prefix'], function (RouteBuilder $routes) {
    // As rotas conectadas aqui são prefixadas com '/my_prefix'
    $routes->connect('/:controller');
});
```

Você também pode definir prefixos dentro dos escopos de plugins:

```
Router::plugin('DebugKit', function ($routes) {
    $routes->prefix('Admin', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

O exemplo acima criaria um modelo de rota como `/debug_kit/admin/:controller`. A rota conectada teria os elementos de rota `plugin` e `prefix` definidos.

Ao definir prefixos, você pode aninhar vários prefixos, se necessário:

```
Router::prefix('Manager', function ($routes) {
    $routes->prefix('Admin', function ($routes) {
        $routes->connect('/:controller/:action');
    });
});
```

O exemplo acima, criaria um modelo de rota como `/manager/admin/:controller/:action`. A rota conectada teria o elemento de rota `prefix` configurado como `Manager/Admin`.

O prefixo atual estará disponível nos métodos do controlador através de `$this->request->getParam('prefix')`.

Ao usar rotas de prefixo, é importante definir a opção `prefix` e usar o mesmo formato de camelo que é usado no método `prefix()`. Veja como criar esse link usando o HTML Helper:

```
// Entre em uma rota prefixada.
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'Manager/Admin', 'controller' => 'Articles', 'action' => 'add']
);

// Deixe um prefixo
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Nota: Você deve conectar rotas de prefixo *antes* de conectar rotas de fallback.

Criando links para rotas de prefixo

Você pode criar links que apontam para um prefixo, adicionando a chave de prefixo à sua matriz de URL:

```
echo $this->Html->link(
    'New admin todo',
    ['prefix' => 'Admin', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Ao usar o aninhamento, você precisa encadeá-los:

```
echo $this->Html->link(
    'New todo',
    ['prefix' => 'Admin/MyPrefix', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Isso vincularia a um controlador com o namespace `App\\Controller\\Admin\\MyPrefix` e o caminho do arquivo `src/Controller/Admin/MyPrefix/TodoItemsController.php`.

Nota: O prefixo é sempre camel case aqui, mesmo que o resultado do roteamento seja tracejado. A própria rota fará a inflexão, se necessário.

Roteamento de Plugins

static Cake\Routing\Router::**plugin**(*$name*, *$options = []*, *$callback*)

As rotas para *Plugins* devem ser criadas usando o método `plugin()`. Este método cria um novo escopo de roteamento para as rotas do plugin:

```
Router::plugin('DebugKit', function ($routes) {
    // As rotas conectadas aqui são prefixadas com '/debug_kit' e
    // têm o elemento de rota do plug-in definido como 'DebugKit'.
    $routes->connect('/:controller');
});
```

Ao criar escopos de plug-in, você pode personalizar o caminho usando a opção `path`:

```
Router::plugin('DebugKit', ['path' => '/debugger'], function ($routes) {
    // As rotas conectadas aqui são prefixadas com '/debug_kit' e
    // têm o elemento de rota do plug-in definido como 'DebugKit'.
    $routes->connect('/:controller');
});
```

Ao usar escopos, você pode aninhar escopos de plug-ins dentro de escopos de prefixo:

```
Router::prefix('Admin', function ($routes) {
    $routes->plugin('DebugKit', function ($routes) {
        $routes->connect('/:controller');
    });
});
```

O exemplo acima criaria uma rota parecida com `/admin/debug_kit/:controller`. Teria o conjunto de elementos de rota `prefix` e `plugin`. A seção *Rotas para Plugin* possui mais informações sobre a construção de rotas para plugins.

Criando links para rotas de plugins

Você pode criar links que apontam para um plug-in, adicionando a chave do plug-in a seu array de URL:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => 'Todo', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Por outro lado, se a solicitação ativa for uma solicitação de plug-in e você desejar criar um link que não possua plug-in, faça o seguinte:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => null, 'controller' => 'Users', 'action' => 'profile']
);
```

Ao definir `'plugin' => null`, você diz ao roteador que deseja criar um link que não faça parte de um plug-in.

Roteamento otimizado para SEO

Alguns desenvolvedores preferem usar hífens nos URLs, pois é percebido que eles fornecem melhores classificações nos mecanismos de pesquisa. A classe `DashedRoute` pode ser usada em seu aplicativo com a capacidade de rotear nomes de plugins, controladores e ações «camelizadas» para uma URL tracejada.

Por exemplo, se tivéssemos um plugin `ToDo`, com um controlador `TodoItems` e uma ação `showItems()`, ele poderia ser acessado em `/to-do/todo-items/show-items` com a seguinte conexão do roteador:

```
use Cake\Routing\Route\DashedRoute;

Router::plugin('ToDo', ['path' => 'to-do'], function ($routes) {
    $routes->fallbacks(DashedRoute::class);
});
```

Correspondendo a métodos HTTP específicos

As rotas podem corresponder a métodos HTTP específicos usando os métodos auxiliares de verbo HTTP:

```
Router::scope('/', function($routes) {
    // Esta rota corresponde apenas às solicitações POST.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Corresponder vários verbos
    // Antes do 3.5, use $options['_method'] para definir o método
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
    )->setMethods(['POST', 'PUT']);
});
```

Você pode combinar vários métodos HTTP usando uma matriz. Como o parâmetro `_method` é uma chave de roteamento, ele participa da análise e geração de URLs. Para gerar URLs para rotas específicas de métodos, você precisará incluir a chave `_method` ao gerar a URL:

```
$url = Router::url([
    'controller' => 'Reviews',
    'action' => 'start',
    '_method' => 'POST',
]);
```

Nomes de host específicos correspondentes

As rotas podem usar a opção `_host` para corresponder apenas a hosts específicos. Você pode usar o curinga `*.` para corresponder a qualquer subdomínio:

```
Router::scope('/', function($routes) {
    // Esta rota corresponde apenas a http://images.example.com
    // Antes da versão 3.5, use a opção _host
    $routes->connect(
        '/images/default-logo.png',
        ['controller' => 'Images', 'action' => 'default']
    )->setHost('images.example.com');

    // Esta rota corresponde apenas a http://*.example.com
    $routes->connect(
        '/images/old-logo.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('*.example.com');
});
```

A opção `_host` também é usada na geração de URL. Se a opção `_host` especificar um domínio exato, esse domínio

será incluído no URL gerado. No entanto, se você usar um curinga, precisará fornecer o parâmetro `_host` ao gerar URLs:

```
// Se você tem esta rota
$routes->connect(
    '/images/old-logo.png',
    ['controller' => 'Images', 'action' => 'oldLogo']
)->setHost('images.example.com');

// Você precisa disso para gerar um URL
echo Router::url([
    'controller' => 'Images',
    'action' => 'oldLogo',
    '_host' => 'images.example.com',
]);
```

Extensões de arquivo de roteamento

static Cake\Routing\Router::**extensions**(*string|array|null $extensions*, *$merge = true*)

Para lidar com diferentes extensões de arquivo com suas rotas, você pode definir extensões em nível global e de escopo. A definição de extensões globais pode ser obtida através do método estático *Router::extensions()* dos roteadores:

```
Router::extensions(['json', 'xml']);
// ...
```

Isso afetará **todas** as rotas que serão conectadas **posteriormente**, independentemente do seu escopo.

Para restringir extensões a escopos específicos, você pode defini-las usando o método `Cake\Routing\RouteBuilder::setExtensions()`:

```
Router::scope('/', function ($routes) {
    // Prior to 3.5.0 use `extensions()`
    $routes->setExtensions(['json', 'xml']);
});
```

Isso habilitará as extensões nomeadas para todas as rotas que estão sendo conectadas esse escopo **após** a chamada de `setExtensions()`, incluindo aqueles que estão sendo conectado em escopos aninhados. Semelhante ao método global *Router::extensions()*, quaisquer rotas conectadas antes da chamada não herdarão as extensões.

Nota: A configuração das extensões deve ser a primeira coisa que você faz em um escopo, pois as extensões serão aplicadas apenas às rotas conectadas **depois** que as extensões forem definidas.

Lembre-se também de que escopos reabertos **não** herdarão extensões definidas em escopos abertos anteriormente.

Ao usar extensões, você diz ao roteador para remover as extensões de arquivo correspondentes e analisar o que resta. Se você deseja criar uma URL como /page/title-of-page.html, crie sua rota usando:

```
Router::scope('/page', function ($routes) {
    // Antes de 3.5.0 use `extensions()`
    $routes->setExtensions(['json', 'xml', 'html']);
    $routes->connect(
        '/:title',
```

(continues on next page)

(continuação da página anterior)

```
        ['controller' => 'Pages', 'action' => 'view']
    )->setPass(['title']);
});
```

Para criar links que mapeiam de volta para as rotas, basta usar:

```
$this->Html->link(
    'Link title',
    ['controller' => 'Pages', 'action' => 'view', 'title' => 'super-article', '_ext' =>
↪'html']
);
```

As extensões de arquivo são usadas por *Request Handler (Tratamento de Requisições)* para fazer a troca automática de exibição com base nos tipos de conteúdo.

Conectando Middleware com Escopo

Embora o Middleware possa ser aplicado a todo o aplicativo, a aplicação do middleware a escopos de roteamento específicos oferece mais flexibilidade, pois você pode aplicar o middleware apenas onde for necessário, permitindo que o middleware não se preocupe com como/onde está sendo aplicado.

Antes que o middleware possa ser aplicado a um escopo, ele precisa ser registrado na coleção de rotas:

```
// Em 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());
});
```

Uma vez registrado, o middleware com escopo definido pode ser aplicado a escopos específicos:

```
$routes->scope('/cms', function ($routes) {
    // Habilita os middlewares de CSRF & cookies
    $routes->applyMiddleware('csrf', 'cookies');
    $routes->get('/articles/:action/*', ['controller' => 'Articles']);
});
```

Nas situações em que você tem escopos aninhados, os escopos internos herdarão o middleware aplicado no escopo que o contém:

```
$routes->scope('/api', function ($routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function ($routes) {
        $routes->applyMiddleware('v1compat');
        // Define routes here.
    });
});
```

No exemplo acima, as rotas definidas em `/v1` terão os middlewares “ratelimit”, “auth.api” e “v1compat” aplicados. Se você reabrir um escopo, o middleware aplicado às rotas em cada escopo será isolado:

```
$routes->scope('/blog', function ($routes) {
    $routes->applyMiddleware('auth');
    // Conecte as ações autenticadas para o blog aqui.
});
$routes->scope('/blog', function ($routes) {
    // Conecte as ações públicas para o blog aqui.
});
```

No exemplo acima, os dois usos do escopo `/blog` não compartilham middleware. No entanto, esses dois escopos herdarão o middleware definido em seus escopos anexos.

Agrupando Middlewares

Para ajudar a manter o seu código de rota DRY (Do not Repeat Yourself) o middleware pode ser combinado em grupos. Uma vez que grupos combinados podem ser aplicados, como o middleware:

```
$routes->registerMiddleware('cookie', new EncryptedCookieMiddleware());
$routes->registerMiddleware('auth', new AuthenticationMiddleware());
$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->middlewareGroup('web', ['cookie', 'auth', 'csrf']);

// Aplica o grupo
$routes->applyMiddleware('web');
```

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 queríamos permitir acesso REST a um controlador de receita, faríamos algo como esta:

```
// no arquivo config/routes.php...

Router::scope('/', function ($routes) {
    // anterior a versao 3.5.0 usar `extensions()`
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

A primeira linha configura uma série de rotas padrão para REST, de 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. A váriavel `_method` POST
2. O `X_HTTP_METHOD_OVERRIDE`
3. O cabeçalho `REQUEST_METHOD`

A váriavel `_method` POST é útil na quando há um navegador como 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.

Criando rotas de recursos aninhados

Depois de conectar recursos em um escopo, você também pode conectar rotas para sub-recursos. As rotas de sub-recursos serão precedidas pelo nome do recurso original e um parâmetro `id`. Por exemplo:

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments');
    });
});
```

Irá gerar rotas de recursos para `articles` e `comments`. As rotas de comentários terão a aparência de:

```
/api/articles/:article_id/comments
/api/articles/:article_id/comments/:id
```

Você pode obter o `article_id` em `CommentsController` por:

```
$this->request->getParam('article_id');
```

Por padrão, as rotas de recursos são mapeadas para o mesmo prefixo que o escopo que contém. Se você tiver controladores de recursos aninhados e não aninhados, poderá usar um controlador diferente em cada contexto usando prefixos:

```
Router::scope('/api', function ($routes) {
    $routes->resources('Articles', function ($routes) {
        $routes->resources('Comments', ['prefix' => 'Articles']);
    });
});
```

A descrição acima mapeia o recurso "Comments" para `App\Controller\Articles\CommentsController`. Ter controladores separados permite manter a lógica do controlador mais simples. Os prefixos criados dessa maneira são compatíveis com *Prefix Routing*.

Nota: Embora você possa aninhar recursos tão profundamente quanto necessário, não é recomendável aninhar mais de 2 recursos juntos.

Limitando as rotas criadas

Por padrão, o CakePHP conectará seis rotas para cada recurso. Se você deseja conectar apenas rotas de recursos específicos, use a opção `only`:

```
$routes->resources('Articles', [
    'only' => ['index', 'view']
]);
```

Criaria rotas de recurso somente leitura. Os nomes das rotas são `create`, `update`, `view`, `index` e `delete`

Alterando as ações usadas no controlador

Pode ser necessário alterar os nomes de ação do controlador usados ao conectar rotas. Por exemplo, se sua ação `edit()` é chamada `put()`, você pode usar a chave `actions` para renomear as ações usadas:

```
$routes->resources('Articles', [
    'actions' => ['update' => 'put', 'create' => 'add']
]);
```

O exemplo acima usaria `put()` para a ação `edit()` e `add()` em vez de `create()`.

Mapeando rotas de recursos adicionais

Você pode mapear métodos de recursos adicionais usando a opção `map`:

```
$routes->resources('Articles', [
   'map' => [
       'deleteAll' => [
           'action' => 'deleteAll',
           'method' => 'DELETE'
       ]
   ]
]);
// Isso conectaria a /articles/deleteAll
```

Além das rotas padrão, isso também conectaria uma rota para */articles/delete_all*. Por padrão, o segmento do caminho corresponderá ao nome da chave. Você pode usar a chave "path" dentro da definição de recurso para personalizar o nome do caminho:

```
$routes->resources('Articles', [
    'map' => [
        'updateAll' => [
            'action' => 'updateAll',
            'method' => 'DELETE',
            'path' => '/update_many'
        ],
    ]
]);
// Isso conectaria a /articles/update_many
```

Se você definir "only" e "map", verifique se seus métodos mapeados também estão na lista "only".

Classes de rota personalizadas para rotas de recursos

Você pode fornecer a chave `connectOptions` na matriz `$options` para `resources()` para fornecer configurações personalizadas usadas por `connect()`:

```
Router::scope('/', function ($routes) {
    $routes->resources('Books', [
        'connectOptions' => [
            'routeClass' => 'ApiRoute',
        ]
    ];
});
```

Inflexão de URL para rotas de recursos

Por padrão, os fragmentos de URL dos controladores com várias palavras são a forma sublinhada do nome do controlador. Por exemplo, fragmento de URL do `BlogPostsController` seria **/blog_posts**.

Você pode especificar um tipo de inflexão alternativo usando a opção `inflect`:

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'dasherize' // Will use ``Inflector::dasherize()``
    ]);
});
```

O exemplo acima irá gerar URLs com estilo semelhantes a: **/blog-posts**.

Nota: A partir do CakePHP 3.1, o esqueleto oficial do aplicativo usa `DashedRoute` como sua classe de rota padrão. Recomenda-se o uso da opção `'inflect' => 'dasherize'` ao conectar rotas de recursos para garantir a consistência da URL

Alterando o elemento de caminho

Por padrão, as rotas de recursos usam um formulário flexionado do nome do recurso para o segmento de URL. Você pode definir um segmento de URL personalizado com a opção `path`:

```
Router::scope('/', function ($routes) {
    $routes->resources('BlogPosts', ['path' => 'posts']);
});
```

Passando Argumentos

Os argumentos passados são argumentos adicionais ou segmentos de caminho que são usados ao fazer uma solicitação. Eles são frequentemente usados para passar parâmetros para os métodos do seu controlador:

```
http://localhost/calendars/view/recent/mark
```

No exemplo acima, os argumentos `recent` e `mark` são passados para `CalendarsController::view()`. Os argumentos passados são fornecidos aos seus controladores de três maneiras. Primeiro, como argumentos para o método de ação chamado, segundo, eles estão disponíveis em `$this->request->getParam('pass')` como uma matriz numerada indexada. Ao usar rotas personalizadas, você pode forçar parâmetros específicos para entrar e os argumentos passados também.

Se você visitar o URL mencionado anteriormente, e teve uma ação de controlador que se parecia com:

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
        debug(func_get_args());
    }
}
```

Você obteria a seguinte saída:

```
Array
(
    [0] => recent
    [1] => mark
)
```

Esses mesmos dados também estão disponíveis em `$this->request->getParam('pass')` em seus controladores, views e auxiliares. Os valores na matriz de `pass` são indexados numericamente com base na ordem em que aparecem no URL chamado:

```
debug($this->request->getParam('pass'));
```

Qualquer um dos itens acima produziria:

```
Array
(
    [0] => recent
    [1] => mark
)
```

Ao gerar URLs, usando a :term: *routing array*, você adiciona argumentos passados como valores sem chaves de string na matriz:

```
['controller' => 'Articles', 'action' => 'view', 5]
```

Como 5 tem uma chave numérica, ela é tratada como um argumento passado.

Gerando URLs

static Cake\Routing\Router::**url**(*$url = null*, *$full = false*)

Gerar URLs ou roteamento reverso é um recurso do CakePHP que é usado para permitir que você altere sua estrutura de URLs sem precisar modificar todo o seu código. Usando *routing arrays* para definir seus URLs, você poderá configurar rotas posteriormente e os URLs gerados serão atualizados automaticamente.

Se você criar URLs usando strings como:

```
$this->Html->link('View', '/articles/view/' . $id);
```

E depois decida que `/articles` deve realmente ser chamado de "posts", você precisará passar por todo o aplicativo renomeando URLs. No entanto, se você definiu seu link como:

```
$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

Então, quando você decidiu alterar seus URLs, pode fazê-lo definindo uma rota. Isso alteraria o mapeamento de URLs recebidos, bem como os URLs gerados.

Ao usar URLs de matriz, você pode definir parâmetros de sequência de consulta e fragmentos de documento usando chaves especiais:

```
Router::url([
    'controller' => 'Articles',
    'action' => 'index',
    '?' => ['page' => 1],
    '#' => 'top'
]);

// Irá gerar uma URL como.
/articles/index?page=1#top
```

O roteador também converterá quaisquer parâmetros desconhecidos em uma matriz de roteamento em parâmetros de querystring. O `?` É oferecido para compatibilidade com versões anteriores do CakePHP.

Você também pode usar qualquer um dos elementos de rota especiais ao gerar URLs:

- `_ext` Usado para *Extensões de arquivo de roteamento* roteamento.
- `_base` define como `false` para remover o caminho base da URL gerada. Se seu aplicativo não estiver no diretório raiz, isso poderá ser usado para gerar URLs que são "relativos ao cake".
- `_scheme` Configure para criar links em diferentes esquemas, como `webcal` ou `ftp`. O padrão é o esquema atual.
- `_host` Defina o host a ser usado para o link. O padrão é o host atual.
- `_port` Defina a porta se precisar criar links em portas não padrão.
- `_method` Defina o verbo HTTP para o qual a URL é.
- `_full` Se `true`, a constante `FULL_BASE_URL` será anexada aos URLs gerados.
- `_https` Defina como `true` para converter o URL gerado em https ou `false` para forçar http.
- `_name` Nome da rota. Se você configurou rotas nomeadas, poderá usar esta chave para especificá-la.

Rotas de redirecionamento

O roteamento de redirecionamento permite emitir redirecionamentos de status HTTP 30x para rotas de entrada e apontá-los para URLs diferentes. Isso é útil quando você deseja informar aos aplicativos clientes que um recurso foi movido e não deseja expor dois URLs para o mesmo conteúdo.

As rotas de redirecionamento são diferentes das rotas normais, pois executam um redirecionamento de cabeçalho real se uma correspondência for encontrada. O redirecionamento pode ocorrer para um destino dentro do seu aplicativo ou para um local externo:

```
Router::scope('/', function ($routes) {
    $routes->redirect(
        '/home/*',
        ['controller' => 'Articles', 'action' => 'view'],
        ['persist' => true]
        // Ou ['persist' => ['id']] para roteamento padrão em que
        // a ação de exibição espera o $id como argumento.
    );
})
```

Redireciona `/home/*` para `/articles/view` e passa os parâmetros para `/articles/view`. O uso de uma matriz como destino de redirecionamento permite usar outras rotas para definir para onde uma string de URL deve ser redirecionada. Você pode redirecionar para locais externos usando URLs de string como destino:

```
Router::scope('/', function ($routes) {
    $routes->redirect('/articles/*', 'https://google.com', ['status' => 302]);
});
```

Isso redirecionaria `/articles/*` para `https://google.com` com um status HTTP 302.

Classes de rota personalizadas

As classes de rota personalizadas permitem estender e alterar como rotas individuais analisam solicitações e manipulam o roteamento reverso. As classes de rota têm algumas convenções:

- As classes de rota devem ser encontradas no espaço de nome `Routing\\Route` do seu aplicativo ou plugin.
- As classes de rota devem estender `Cake\Routing\Route`.
- As classes de rota devem implementar os métodos `match()` e/ou `parse()`.

O método `parse()` é usado para analisar uma URL recebida. Ele deve gerar uma matriz de parâmetros de solicitação que podem ser resolvidos em um controlador e ação. Retorne `false` deste método para indicar uma falha na correspondência.

O método `match()` é usado para corresponder a uma matriz de parâmetros de URL e criar uma URL de string. Se os parâmetros de URL não corresponderem à rota, `false` deve ser retornado.

Você pode usar uma classe de rota personalizada ao fazer uma rota usando a opção `routeClass`:

```
$routes->connect(
     '/:slug',
     ['controller' => 'Articles', 'action' => 'view'],
     ['routeClass' => 'SlugRoute']
);
```

(continues on next page)

(continuação da página anterior)

```
// Ou configurando o routeClass no seu escopo.
$routes->scope('/', function ($routes) {
    // Antes de 3.5.0 use `routeClass()`
    $routes->setRouteClass('SlugRoute');
    $routes->connect(
        '/:slug',
        ['controller' => 'Articles', 'action' => 'view']
    );
});
```

Esta rota criaria uma instância de `SlugRoute` e permitiria a você implementar a manipulação de parâmetros personalizados. Você pode usar as classes de rota do plugin usando standard:term:*sintaxe plugin*.

Classe de rota padrão

`static` Cake\Routing\Router::**defaultRouteClass**(*$routeClass = null*)

Se você deseja usar uma classe de rota alternativa para todas as suas rotas além do padrão `Route`, pode fazê-lo chamando `Router::defaultRouteClass()` antes de configurar qualquer rota e evitar especificar a opção `routeClass` para cada rota. Por exemplo, usando:

```
use Cake\Routing\Route\InflectedRoute;

Router::defaultRouteClass(InflectedRoute::class);
```

fará com que todas as rotas conectadas depois disso usem a classe de rota `InflectedRoute`. Chamar o método sem um argumento retornará a classe de rota padrão atual.

Método de fallbacks

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

O método de fallbacks é um atalho simples para definir rotas padrão. O método usa a classe de roteamento passada para as regras definidas ou, se nenhuma classe for fornecida, a classe retornada por `Router::defaultRouteClass()` será usada.

Chamando fallbacks assim:

```
use Cake\Routing\Route\DashedRoute;

$routes->fallbacks(DashedRoute::class);
```

É equivalente às seguintes chamadas explícitas:

```
use Cake\Routing\Route\DashedRoute;

$routes->connect('/:controller', ['action' => 'index'], ['routeClass' =>␣
```

(continues on next page)

(continuação da página anterior)

```
↪DashedRoute::class]);
$routes->connect('/:controller/:action/*', [], ['routeClass' => DashedRoute::class]);
```

Nota: O uso da classe de rota padrão (`Route`) com fallbacks ou qualquer rota com elementos de rota `:plugin` e/ou `:controller` resultará em uma URL inconsistente

Criando parâmetros de URL persistentes

Você pode se conectar ao processo de geração de URL usando as funções de filtro de URL. As funções de filtro são chamadas *antes* dos URLs corresponderem às rotas, permitindo preparar os URLs antes do roteamento.

As funções de filtro de retorno de chamada devem esperar os seguintes parâmetros:

- `$params` Os parâmetros de URL que estão sendo processados.
- `$request` A solicitação atual.

A função de filtro de URL deve *sempre* retornar os parâmetros, mesmo que não seja modificada.

Os filtros de URL permitem implementar recursos como parâmetros persistentes:

```
Router::addUrlFilter(function ($params, $request) {
    if ($request->getParam('lang') && !isset($params['lang'])) {
        $params['lang'] = $request->getParam('lang');
    }

    return $params;
});
```

As funções de filtro são aplicadas na ordem em que estão conectadas.

Outro caso de uso está mudando uma determinada rota no tempo de execução (rotas de plug-in, por exemplo):

```
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;
});
```

Isso alterará a seguinte rota:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Languages', 'action' => 'view', 'es
↪']);
```

nisso:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Locations', 'action' => 'index',
↪'language' => 'es']);
```

Manipulando parâmetros nomeados em URLs

Embora os parâmetros nomeados tenham sido removidos no CakePHP 3.0, os aplicativos podem ter URLs publicados que os contêm. Você pode continuar aceitando URLs contendo parâmetros nomeados.

No método `beforeFilter()` do seu controlador, você pode chamar `parseNamedParams()` para extrair qualquer parâmetro nomeado dos argumentos passados:

```
public function beforeFilter(Event $event)
{
    parent::beforeFilter($event);
    Router::parseNamedParams($this->request);
}
```

Isso preencherá `$this->request->getParam('named')` com quaisquer parâmetros nomeados encontrados nos argumentos passados. Qualquer argumento passado que foi interpretado como um parâmetro nomeado será removido da lista de argumentos passados.

CAPÍTULO 9

Objetos de Requisição e Resposta

Os objetos de solicitação e resposta fornecem uma abstração em torno de solicitações e respostas HTTP. O objeto de solicitação no CakePHP permite que você examine uma solicitação de entrada, enquanto o objeto de resposta permite criar respostas HTTP sem esforço do seus controladores.

Requisição

class Cake\Http**ServerRequest**

`ServerRequest` é o objeto de solicitação padrão usado no CakePHP. Ele centraliza vários recursos para interrogar e interagir com os dados da solicitação. Em cada solicitação, uma requisição é criada e depois passada por referência às várias camadas de um aplicativo que usam dados da solicitação. Por padrão, a solicitação é atribuída a `$this->request` e está disponível em Controllers, Cells, Views e Helpers. Você também pode acessá-lo em Components usando a referência do controlador. Algumas das tarefas que o `ServerRequest` executa incluem:

- Processar as matrizes GET, POST e FILES nas estruturas de dados que você conhece.
- Fornecer introspecção do ambiente referente à solicitação. Informações como os cabeçalhos enviados, o endereço IP do cliente e os nomes de subdomínio/domínio no servidor em que seu aplicativo está sendo executado.
- Fornecendo acesso a parâmetros de solicitação, como índices de matriz e propriedades de objetos.

O objeto de solicitação do CakePHP implementa a PSR-7 ServerRequestInterface[63] facilitando o uso de bibliotecas de fora do CakePHP.

[63] https://www.php-fig.org/psr/psr-7/

Parâmetros de Requsição

A solicitação expõe parâmetros de roteamento através do método `getParam()`:

```
$controllerName = $this->request->getParam('controller');
```

Para obter todos os parâmetros de roteamento como uma matriz, use `getAttribute()`:

```
$parameters = $this->request->getAttribute('params');
```

Todos *Elementos de Rota* são acessados através desta interface.

Além de *Elementos de Rota*, você também precisa frequentemente acessar *Passando Argumentos*. Ambos estão disponíveis no objeto de solicitação também:

```
// Argumentos passados
$passedArgs = $this->request->getParam('pass');
```

Todos fornecerão acesso aos argumentos passados. Existem vários parâmetros importantes/úteis que o CakePHP usa internamente, todos eles também são encontrados nos parâmetros de roteamento:

- `plugin` O plug-in que manipula a solicitação. Será nulo quando não houver plug-in.
- `controller` O controlador que manipula a solicitação atual.
- `action` A ação que manipula a solicitação atual.
- `prefix` O prefixo da ação atual. Veja *Prefix Routing* para mais informações.

Parâmetros em URL

`Cake\Http\ServerRequest::getQuery`(*$name*, *$default = null*)

Os parâmetros em URL podem ser lidos usando o método `getQuery()`:

```
// A URL é /posts/index?page=1&sort=title
$page = $this->request->getQuery('page');
```

Você pode acessar diretamente a propriedade query, ou pode usar o método `getQuery()` para ler a matriz de consultas de URL de maneira livre de erros. Quaisquer chaves que não existirem retornarão `null`:

```
$foo = $this->request->getQuery('value_that_does_not_exist');
// $foo === null

// Você também pode fornecer valores padrão
$foo = $this->request->getQuery('does_not_exist', 'default val');
```

Se você deseja acessar todos os parâmetros da consulta, pode usar `getQueryParams()`:

```
$query = $this->request->getQueryParams();
```

Dados do Corpo da Requisição

Cake\Http\ServerRequest::**getData**(*$name*, *$default = null*)

Todos os dados do POST podem ser acessados usando *Cake\Http\ServerRequest::getData()*. Qualquer dado de formulário que contenha um prefixo `data` terá esse prefixo de dados removido. Por exemplo:

```
// Uma entrada com um atributo de nome igual a 'MyModel [title]' está acessível em
$title = $this->request->getData('MyModel.title');
```

Quaisquer chaves que não existem retornarão `null`:

```
$foo = $this->request->getData('Value.that.does.not.exist');
// $foo == null
```

Dados PUT, PATCH ou DELETE

Cake\Http\ServerRequest::**input**(*$callback*[, *$options*])

Ao criar serviços REST, você geralmente aceita dados de solicitação em solicitações `PUT` e `DELETE`. Qualquer dado do corpo da solicitação `application/x-www-form-urlencoded` será automaticamente analisado e definido como `$this->data` para as solicitações `PUT` e `DELETE`. Se você estiver aceitando dados JSON ou XML, veja abaixo como acessar esses corpos de solicitação.

Ao acessar os dados de entrada, você pode decodificá-los com uma função opcional. Isso é útil ao interagir com o conteúdo do corpo da solicitação XML ou JSON. Parâmetros adicionais para a função de decodificação podem ser passados como argumentos para `input()`:

```
$jsonData = $this->request->input('json_decode');
```

Variáveis de Ambiente ($_SERVER e $_ENV)

Cake\Http\ServerRequest::**env**(*$key*, *$value = null*)

`ServerRequest::env()` é um wrapper para a função global `env()` e atua como um getter/setter para variáveis de ambiente sem precisar modificar as globais `$_SERVER` e `$_ENV`:

```
// Obter o host
$host = $this->request->env('HTTP_HOST');

// Defina um valor, geralmente útil nos testes.
$this->request->env('REQUEST_METHOD', 'POST');
```

Para acessar todas as variáveis de ambiente em uma solicitação, use `getServerParams()`:

```
$env = $this->request->getServerParams();
```

Dados XML ou JSON

Os aplicativos que empregam *REST* geralmente trocam dados em corpos de postagem não codificados em URL. Você pode ler dados de entrada em qualquer formato usando *input()*. Ao fornecer uma função de decodificação, você pode receber o conteúdo em um formato desserializado:

```
// Obter dados codificados em JSON enviados para uma ação PUT/POST
$jsonData = $this->request->input('json_decode');
```

Alguns métodos de desserialização requerem parâmetros adicionais quando chamados, como o parâmetro "as array" em `json_decode`. Se você desejar que o XML seja convertido em um objeto DOMDocument, *input()* também suporta a passagem de parâmetros adicionais:

```
// Obter dados codificados em XML enviados para uma ação PUT/POST
$data = $this->request->input('Cake\Utility\Xml::build', ['return' => 'domdocument']);
```

Informações de Caminho

O objeto de solicitação também fornece informações úteis sobre os caminhos em seu aplicativo. Os atributos `base` e `webroot` são úteis para gerar URLs e determinar se seu aplicativo está ou não em um subdiretório. Os atributos que você pode usar são:

```
// Suponha que o URL da solicitação atual seja /subdir/articles/edit/1?page=1

// Possui /subdir/articles/edit/1?page=1
$here = $request->getRequestTarget();

// Possui /subdir
$base = $request->getAttribute('base');

// Possui /subdir/
$base = $request->getAttribute('webroot');
```

Verificando as Condições da Solicitação

`Cake\Http\ServerRequest::is`(*$type*, *$args...*)

O objeto de solicitação fornece uma maneira fácil de inspecionar determinadas condições em uma determinada solicitação. Usando o método `is()`, você pode verificar várias condições comuns, bem como inspecionar outros critérios de solicitação específicos do aplicativo:

```
$isPost = $this->request->is('post');
```

Você também pode estender os detectores de solicitação disponíveis, usando *Cake\Http\ServerRequest::addDetector()* para criar novos tipos de detectores. Existem quatro tipos diferentes de detectores que você pode criar:

- Comparação de valores do ambiente - Compara um valor obtido de *env()* para igualdade com o valor fornecido.
- Comparação de valores padrão - A comparação de valores padrão permite comparar um valor obtido de *env()* com uma expressão regular.

- Comparação baseada em opção - Comparações baseadas em opção usam uma lista de opções para criar uma expressão regular. As chamadas subseqüentes para adicionar um detector de opções já definido mesclarão as opções.
- Detectores de retorno de chamada - Os detectores de retorno de chamada permitem que você forneça um tipo de "callback" para lidar com a verificação. O retorno de chamada receberá o objeto de solicitação como seu único parâmetro.

`Cake\Http\ServerRequest::addDetector`(*$name*, *$options*)

Alguns exemplos seriam:

```
// Adicione um detector de ambiente.
$this->request->addDetector(
    'post',
    ['env' => 'REQUEST_METHOD', 'value' => 'POST']
);

// Adicione um detector de valor padrão.
$this->request->addDetector(
    'iphone',
    ['env' => 'HTTP_USER_AGENT', 'pattern' => '/iPhone/i']
);

// Adicione um detector de opção
$this->request->addDetector('internalIp', [
    'env' => 'CLIENT_IP',
    'options' => ['192.168.0.101', '192.168.0.100']
]);

// Adicione um detector de callback. Deve ser uma chamada válida.
$this->request->addDetector(
    'awesome',
    function ($request) {
        return $request->getParam('awesome');
    }
);

// Adicione um detector que use argumentos adicionais.
$this->request->addDetector(
    'controller',
    function ($request, $name) {
        return $request->getParam('controller') === $name;
    }
);
```

`Request` também inclui métodos como *Cake\Http\ServerRequest::domain()*, *Cake\Http\ServerRequest::subdomains()* e *Cake\Http\ServerRequest::host()* para ajudar aplicativos com sub-domínios, tenha uma vida um pouco mais fácil.

Existem vários detectores embutidos que você pode usar:

- `is('get')` Verifique se a solicitação atual é um GET.
- `is('put')` Verifique se a solicitação atual é um PUT.
- `is('patch')` Verifique se a solicitação atual é um PATCH.

- `is('post')` Verifique se a solicitação atual é um POST.
- `is('delete')` Verifique se a solicitação atual é um DELETE.
- `is('head')` Verifique se a solicitação atual é HEAD.
- `is('options')` Verifique se a solicitação atual é OPTIONS.
- `is('ajax')` Verifique se a solicitação atual veio com X-Requested-With = XMLHttpRequest.
- `is('ssl')` Verifique se a solicitação é via SSL.
- `is('flash')` Verifique se a solicitação possui um User-Agent de Flash.
- `is('requested')` Verifique se a solicitação possui um parâmetro de consulta "solicitado" com o valor 1.
- `is('json')` Verifique se a solicitação possui extensão "json" e aceite mimetype "application/json".
- `is('xml')` Verifique se a solicitação possui extensão "xml" e aceite mimetype "application/xml" ou "text/xml".

Dados da Sessão

Para acessar a sessão para uma determinada solicitação, use o método `getSession()` ou use o atributo `session`:

```
$session = $this->request->getSession();
$session = $this->request->getAttribute('session');

$userName = $session->read('Auth.User.name');
```

Para obter mais informações, consulte a documentação *Sessões* para saber como usar o objeto de sessão.

Host e Nome de Domínio

`Cake\Http\ServerRequest::`**`domain`**(*$tldLength = 1*)

Retorna o nome de domínio em que seu aplicativo está sendo executado:

```
// Prints 'example.org'
echo $request->domain();
```

`Cake\Http\ServerRequest::`**`subdomains`**(*$tldLength = 1*)

Retorna os subdomínios em que seu aplicativo está sendo executado como uma matriz:

```
// Retorna ['my', 'dev'] para 'my.dev.example.org'
$subdomains = $request->subdomains();
```

`Cake\Http\ServerRequest::`**`host`**()

Retorna o host em que seu aplicativo está:

```
// Exibe 'my.dev.example.org'
echo $request->host();
```

Lendo o método HTTP

`Cake\Http\ServerRequest::getMethod()`

Retorna o método HTTP com o qual a solicitação foi feita:

```
// Saída POST
echo $request->getMethod();
```

Restringindo Qual Método HTTP Uma Ação Aceita

`Cake\Http\ServerRequest::allowMethod($methods)`

Defina métodos HTTP permitidos. Se não corresponder, lançará `MethodNotAllowedException`. A resposta 405 incluirá o cabeçalho `Allow` necessário com os métodos passados:

```
public function delete()
{
    // Aceite apenas solicitações POST e DELETE
    $this->request->allowMethod(['post', 'delete']);
    ...
}
```

Lendo Cabeçalhos HTTP

Permite acessar qualquer um dos cabeçalhos `HTTP_*` que foram usados para a solicitação. Por exemplo:

```
// Obter o cabeçalho como uma string
$userAgent = $this->request->getHeaderLine('User-Agent');

// Obtenha uma matriz de todos os valores.
$acceptHeader = $this->request->getHeader('Accept');

// Verifique se existe um cabeçalho
$hasAcceptHeader = $this->request->hasHeader('Accept');
```

Enquanto algumas instalações do apache não tornam o cabeçalho `Authorization` acessível, o CakePHP o torna disponível através de métodos específicos do apache, conforme necessário.

`Cake\Http\ServerRequest::referer($local = true)`

Retorna o endereço de referência para a solicitação.

`Cake\Http\ServerRequest::clientIp()`

Retorna o endereço IP do visitante atual.

Confiando em Cabeçalhos de Proxy

Se o seu aplicativo estiver atrás de um balanceador de carga ou em execução em um serviço de nuvem, geralmente você receberá o host, a porta e o esquema do balanceador de carga em suas solicitações. Freqüentemente, os balanceadores de carga também enviam cabeçalhos `HTTP-X-Forwarded-*` com os valores originais. Os cabeçalhos encaminhados não serão usados pelo CakePHP imediatamente. Para que o objeto de solicitação use esses cabeçalhos, defina a propriedade `trustProxy` como `true`:

```
$this->request->trustProxy = true;

// Esses métodos agora usarão os cabeçalhos com proxy.
$port = $this->request->port();
$host = $this->request->host();
$scheme = $this->request->scheme();
$clientIp = $this->request->clientIp();
```

Uma vez que os proxies são confiáveis, o método `clientIp()` usará o *último* endereço IP no cabeçalho `X-Forwarded-For`. Se o seu aplicativo estiver protegido por vários proxies, você poderá usar `setTrustedProxies()` para definir os endereços IP dos proxies em seu controle:

```
$request->setTrustedProxies(['127.1.1.1', '127.8.1.3']);
```

Depois que os proxies forem confiáveis, o `clientIp()` usará o primeiro endereço IP no cabeçalho `X-Forwarded-For`, desde que seja o único valor que não seja de um proxy confiável.

Verificando Aceitar Cabeçalhos

`Cake\Http\ServerRequest::accepts`(*$type = null*)

Descubra quais tipos de conteúdo o cliente aceita ou verifique se ele aceita um tipo específico de conteúdo.

Obter todos os tipos:

```
$accepts = $this->request->accepts();
```

Verifique se há um único tipo:

```
$acceptsJson = $this->request->accepts('application/json');
```

`Cake\Http\ServerRequest::acceptLanguage`(*$language = null*)

Obtenha todos os idiomas aceitos pelo cliente, ou verifique se um idioma específico é aceito.

Obter a lista de idiomas aceitos:

```
$acceptsLanguages = $this->request->acceptLanguage();
```

Verifique se um idioma específico é aceito:

```
$acceptsSpanish = $this->request->acceptLanguage('es-es');
```

Lendo Cookies

Os cookies de solicitação podem ser lidos através de vários métodos:

```
// Obtem o valor de um cookie, ou nulo se o cookie não existir.
$rememberMe = $this->request->getCookie('remember_me');

// Leia o valor ou obtenha o padrão 0
$rememberMe = $this->request->getCookie('remember_me', 0);

// Obter todos os cookies como um hash
$cookies = $this->request->getCookieParams();

// Obter uma instância CookieCollection
$cookies = $this->request->getCookieCollection()
```

Consulte a documentação *Cake\Http\Cookie\CookieCollection* para saber como trabalhar com a coleção de cookies.

Arquivos Enviados

Solicitações expõem os dados do arquivo carregado em `getData()` como matrizes e como objetos `UploadedFileInterface` por `getUploadedFiles()`:

```
// Obter uma lista de objetos UploadedFile
$files = $request->getUploadedFiles();

// Leia os dados do arquivo.
$files[0]->getStream();
$files[0]->getSize();
$files[0]->getClientFileName();

// Move o arquivo.
$files[0]->moveTo($targetPath);
```

Manipulando URIs

Requisições contêm um objeto URI, que tem métodos para interagir com o URI solicitado:

```
// Obtem o URI
$uri = $request->getUri();

// Leia dados fora do URI.
$path = $uri->getPath();
$query = $uri->getQuery();
$host = $uri->getHost();
```

Resposta

class Cake\Http**Response**

Cake\Http\Response é a classe de resposta padrão no CakePHP. Ele encapsula vários recursos e funcionalidades para gerar respostas HTTP em seu aplicativo. Também auxilia nos testes, pois pode ser simulado/esboçado, permitindo que você inspecione os cabeçalhos que serão enviados. Como *Cake\Http\ServerRequest*, *Cake\Http\Response* consolida uma série de métodos encontrados anteriormente em `Controller`, *RequestHandlerComponent* e `Dispatcher`. Os métodos antigos são preteridos no uso de *Cake\Http\Response*.

`Response` fornece uma interface para agrupar tarefas comuns relacionadas à resposta, como:

- Enviar cabeçalhos para redirecionamentos.
- Enviar cabeçalhos de tipo de conteúdo.
- Enviar qualquer cabeçalho.
- Enviar o corpo da resposta.

Lidando com Tipos de Conteúdo

Cake\Http\Response::**withType**(*$contentType = null*)

Você pode controlar o tipo de conteúdo das respostas do seu aplicativo com *Cake\Http\Response::withType()*. Se seu aplicativo precisar lidar com tipos de conteúdo que não estão embutidos no Response, você pode mapeá-los com `type()` também:

```
// Adiciona um tipo de vCard
$this->response->type(['vcf' => 'text/v-card']);

// Defina a resposta Content-Type como vcard
$this->response = $this->response->withType('vcf');
```

Normalmente, você deseja mapear tipos de conteúdo adicionais no retorno de chamada do seu controlador `beforeFilter()`, para poder aproveitar os recursos de troca automática de exibição de *RequestHandlerComponent* se você está usando.

Enviando Arquivos

Cake\Http\Response::**withFile**(*$path*, *$options = []*)

Há momentos em que você deseja enviar arquivos como respostas para suas solicitações. Você pode fazer isso usando *Cake\Http\Response::withFile()*:

```
public function sendFile($id)
{
    $file = $this->Attachments->getFile($id);
    $response = $this->response->withFile($file['path']);
    // Retorna a resposta para impedir que o controlador tente renderizar
    // uma view.
    return $response;
}
```

Como mostrado no exemplo acima, você deve passar o caminho do arquivo para o método. O CakePHP enviará um cabeçalho de tipo de conteúdo adequado se for um tipo de arquivo conhecido listado em *Cake\Http\Response::$_mimeTypes*. Você pode adicionar novos tipos antes de chamar *Cake\Http\Response::withFile()* usando o método *Cake\Http\Response::withType()*.

Se desejar, você também pode forçar o download de um arquivo em vez de ser exibido no navegador, especificando as opções:

```
$response = $this->response->withFile(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);
```

As opções suportadas são:

name
: O nome permite especificar um nome de arquivo alternativo a ser enviado ao usuário.

download
: Um valor booleano indicando se os cabeçalhos devem ser definidos para forçar o download.

Enviando uma String como Arquivo

Você pode responder com um arquivo que não existe no disco, como um pdf ou um ics gerado on-line a partir de uma string:

```
public function sendIcs()
{
    $icsString = $this->Calendars->generateIcs();
    $response = $this->response;

    // Injetar conteúdo da string no corpo da resposta
    $response = $response->withStringBody($icsString);

    $response = $response->withType('ics');

    // Opcionalmente, obriga o download do arquivo
    $response = $response->withDownload('filename_for_download.ics');

    // Retorne o objeto de resposta para impedir que o controlador tente renderizar
    // uma view.
    return $response;
}
```

Os retornos de chamada também podem retornar o corpo como uma sequência:

```
$path = '/some/file.png';
$this->response->body(function () use ($path) {
    return file_get_contents($path);
});
```

Definindo Cabeçalhos

Cake\Http\Response::**withHeader**(*$header*, *$value*)

A configuração dos cabeçalhos é feita com o método *Cake\Http\Response::withHeader()*. Como todos os métodos de interface PSR-7, esse método retorna uma instância *new* com o novo cabeçalho:

```
// Adicionar/substituir um cabeçalho
$response = $response->withHeader('X-Extra', 'My header');

// Define vários cabeçalhos
$response = $response->withHeader('X-Extra', 'My header')
    ->withHeader('Location', 'http://example.com');

// Anexa um valor a um cabeçalho existente
$response = $response->withAddedHeader('Set-Cookie', 'remember_me=1');
```

Os cabeçalhos não são enviados quando definidos. Em vez disso, eles são mantidos até que a resposta seja emitida por `Cake\Http\Server`.

Agora você pode usar o método conveniente `Cake\Http\Response::withLocation()` para definir diretamente ou obter o cabeçalho do local de redirecionamento.

Definindo o Corpo

Cake\Http\Response::**withStringBody**(*$string*)

Para definir uma sequência como o corpo da resposta, faça o seguinte:

```
// Define uma string no corpo da resposta
$response = $response->withStringBody('My Body');

// Se você deseja enviar uma resposta em JSON
$response = $response->withType('application/json')
    ->withStringBody(json_encode(['Foo' => 'bar']));
```

Cake\Http\Response::**withBody**(*$body*)

Para definir o corpo da resposta, use o método `withBody()`, fornecido pelo `Zend\Diactoros\MessageTrait`:

```
$response = $response->withBody($stream);
```

Certifique-se de que `$stream` seja um objeto `Psr\Http\Message\StreamInterface`. Veja abaixo como criar um novo fluxo.

Você também pode transmitir respostas de arquivos usando `Zend\Diactoros\Stream` streams:

```
// Para transmitir a partir de um arquivo
use Zend\Diactoros\Stream;

$stream = new Stream('/path/to/file', 'rb');
$response = $response->withBody($stream);
```

Você também pode transmitir respostas de um retorno de chamada usando o `CallbackStream`. Isso é útil quando você possui recursos como imagens, arquivos CSV ou PDFs que precisam ser transmitidos para o cliente:

```
// Streaming a partir de um retorno de chamada
use Cake\Http\CallbackStream;

// Cria uma imagem
$img = imagecreate(100, 100);
// ...

$stream = new CallbackStream(function () use ($img) {
    imagepng($img);
});
$response = $response->withBody($stream);
```

Definindo o Conjunto de Caracteres

Cake\Http\Response::**withCharset**(*$charset*)

Define o conjunto de caracteres que será usado na resposta:

```
$this->response = $this->response->withCharset('UTF-8');
```

Interagindo com o Cache do Navegador

Cake\Http\Response::**withDisabledCache**()

Às vezes, você precisa forçar os navegadores a não armazenar em cache os resultados de uma ação do controlador. *Cake\Http\Response::withDisabledCache()* é destinado apenas para isso:

```
public function index()
{
    // Desabilita o caching
    $this->response = $this->response->withDisabledCache();
}
```

Aviso: Desativando o armazenamento em cache de domínios SSL ao tentar enviar arquivos no Internet Explorer podem resultar em erros.

Cake\Http\Response::**withCache**(*$since*, *$time = '+1 day'*)

Você também pode dizer aos clientes que deseja que eles armazenem respostas em cache. Usando *Cake\Http\Response::withCache()*:

```
public function index()
{
    // Habilita o caching
    $this->response = $this->response->withCache('-1 minute', '+5 days');
}
```

O exposto acima informava aos clientes para armazenar em cache a resposta resultante por 5 dias, acelerando a experiência dos visitantes. O método `withCache()` define o valor `Last-Modified` para o primeiro argumento. O cabeçalho

`Expires` e a diretiva `max-age` são configurados com base no segundo parâmetro. A diretiva `public` do Cache-Control também é definida.

Ajuste Fino de Cache HTTP

Uma das melhores e mais fáceis maneiras de acelerar seu aplicativo é usar o cache HTTP. Sob esse modelo de armazenamento em cache, você só precisa ajudar os clientes a decidir se devem usar uma cópia em cache da resposta, definindo alguns cabeçalhos, como tempo modificado e tag da entidade de resposta.

Em vez de forçar você a codificar a lógica para armazenar em cache e invalidá-la (atualizando) depois que os dados forem alterados, o HTTP usa dois modelos, expiração e validação, que geralmente são muito mais simples de usar.

Além de usar *Cake\Http\Response::withCache()*, você também pode usar muitos outros métodos para ajustar os cabeçalhos de cache HTTP para tirar proveito do cache do navegador ou do proxy reverso.

O cabeçalho para Controle de Cache

`Cake\Http\Response::`**`withSharable`**(*$public*, *$time = null*)

Usado como modelo de expiração, esse cabeçalho contém vários indicadores que podem alterar a maneira como navegadores ou proxies usam o conteúdo em cache. Um cabeçalho `Cache-Control` pode ser assim:

```
Cache-Control: private, max-age=3600, must-revalidate
```

A classe `Response` ajuda a definir esse cabeçalho com alguns métodos utilitários que produzirão um cabeçalho final `Cache-Control` válido. O primeiro é o método `withSharable()`, que indica se uma resposta deve ser considerada compartilhável entre diferentes usuários ou clientes. Este método realmente controla a parte `public` ou `private` deste cabeçalho. Definir uma resposta como privada indica que a totalidade ou parte dela é destinada a um único usuário. Para tirar proveito dos caches compartilhados, a diretiva de controle deve ser definida como pública.

O segundo parâmetro desse método é usado para especificar uma `idade máxima` para o cache, que é o número de segundos após os quais a resposta não é mais considerada nova:

```
public function view()
{
    // ...
    // Define o controle de cache como público por 3600 segundos
    $this->response = $this->response->withSharable(true, 3600);
}

public function my_data()
{
    // ...
    // Define o Cache-Control como privado por 3600 segundos
    $this->response = $this->response->withSharable(false, 3600);
}
```

`Response` expõe métodos separados para definir cada uma das diretivas no cabeçalho `Cache-Control`.

O Cabeçalho de Expiração

`Cake\Http\Response::withExpires`(*$time*)

Você pode definir o cabeçalho `Expires` para uma data e hora após a qual a resposta não é mais considerada nova. Esse cabeçalho pode ser definido usando o método `withExpires()`:

```
public function view()
{
    $this->response = $this->response->withExpires('+5 days');
}
```

Este método também aceita uma instância `DateTime` ou qualquer string que possa ser analisada pela classe `DateTime`.

O Cabeçalho Etag

`Cake\Http\Response::withEtag`(*$tag*, *$weak = false*)

A validação de cache no HTTP é frequentemente usada quando o conteúdo está em constante mudança e solicita ao aplicativo que gere apenas o conteúdo da resposta se o cache não estiver mais atualizado. Sob esse modelo, o cliente continua a armazenar páginas no cache, mas pergunta sempre ao aplicativo se o recurso foi alterado, em vez de usá-lo diretamente. Isso é comumente usado com recursos estáticos, como imagens e outros assets.

O método `withEtag()` (chamado tag de entidade) é uma string que identifica exclusivamente o recurso solicitado, como a soma de verificação de um arquivo, para determinar se ele corresponde a um recurso em cache.

Para tirar proveito desse cabeçalho, você deve chamar o método `isNotModified()` manualmente ou incluir o seguinte *Request Handler (Tratamento de Requisições)* no seu controlador:

```
public function index()
{
    $articles = $this->Articles->find('all');
    $response = $this->response->withEtag($this->Articles->generateHash($articles));
    if ($response->isNotModified($this->request)) {
        return $response;
    }
    $this->response = $response;
    // ...
}
```

Nota: A maioria dos usuários proxy provavelmente deve considerar o uso do Último Cabeçalho Modificado em vez de Etags por motivos de desempenho e compatibilidade.

O Último Cabeçalho Modificado

Cake\Http\Response::**withModified**(*$time*)

Além disso, no modelo de validação de cache HTTP, você pode definir o cabeçalho `Last-Modified` para indicar a data e a hora em que o recurso foi modificado pela última vez. Definir este cabeçalho ajuda o CakePHP a informar aos clientes de armazenamento em cache se a resposta foi modificada ou não com base em seu cache.

Para tirar proveito desse cabeçalho, você deve chamar o método `isNotModified()` manualmente ou incluir o seguinte *Request Handler (Tratamento de Requisições)* no seu controlador:

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = $this->response->withModified($article->modified);
    if ($response->isNotModified($this->request)) {
        return $response;
    }
    $this->response;
    // ...
}
```

O Cabeçalho Vary

Cake\Http\Response::**withVary**(*$header*)

Em alguns casos, convém veicular conteúdo diferente usando o mesmo URL. Geralmente, esse é o caso se você tiver uma página multilíngue ou responder com HTML diferente, dependendo do navegador. Nessas circunstâncias, você pode usar o cabeçalho `Vary`:

```
$response = $this->response->withVary('User-Agent');
$response = $this->response->withVary('Accept-Encoding', 'User-Agent');
$response = $this->response->withVary('Accept-Language');
```

Enviando Respostas Não Modificadas

Cake\Http\Response::**isNotModified**(*Request $request*)

Compara os cabeçalhos de cache do objeto de solicitação com o cabeçalho de cache da resposta e determina se ele ainda pode ser considerado novo. Nesse caso, exclui o conteúdo da resposta e envia o cabeçalho *304 Not Modified*:

```
// Em um método do controlador.
if ($this->response->isNotModified($this->request)) {
    return $this->response;
}
```

Configurando Cookies

Os cookies podem ser adicionados à resposta usando um array ou um objeto *Cake\Http\Cookie\Cookie*:

```
use Cake\Http\Cookie\Cookie;
use DateTime;

// Adiciona um cookie
$this->response = $this->response->withCookie(new Cookie(
    'remember_me',
    'yes',
    new DateTime('+1 year'), // expiration time
    '/', // path
    '', // domain
    false, // secure
    true // httponly
]);
```

Veja a seção *created-cookies* para saber como usar o objeto cookie. Você pode usar `withExpiredCookie()` para enviar um cookie expirado na resposta. Isso fará com que o navegador remova seu cookie local:

```
$this->response = $this->response->withExpiredCookie('remember_me');
```

Definindo Cabeçalho de Solicitação de Origem Cruzada (CORS)

O método `cors()` é usado para definir o HTTP Access Control[64], são cabeçalhos relacionados com uma interface fluente:

```
$this->response = $this->response->cors($this->request)
    ->allowOrigin(['*.cakephp.org'])
    ->allowMethods(['GET', 'POST'])
    ->allowHeaders(['X-CSRF-Token'])
    ->allowCredentials()
    ->exposeHeaders(['Link'])
    ->maxAge(300)
    ->build();
```

Os cabeçalhos relacionados ao CORS somente serão aplicados à resposta se os seguintes critérios forem atendidos:

1. A solicitação possui um cabeçalho `Origin`.
2. O valor `Origem` da solicitação corresponde a um dos valores de Origin permitidos.

[64] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Erros Comuns com Respostas Imutáveis

Os objetos de resposta oferecem vários métodos que tratam as respostas como objetos imutáveis. Objetos imutáveis ajudam a evitar efeitos colaterais acidentais difíceis de controlar e reduzem os erros causados por chamadas de método causadas pela refatoração dessa ordem de alteração. Embora ofereçam vários benefícios, objetos imutáveis podem levar algum tempo para se acostumar. Qualquer método que comece com `with` opera a resposta de maneira imutável e **sempre** retorna uma **nova** instância. Esquecer de manter a instância modificada é o erro mais frequente que as pessoas cometem ao trabalhar com objetos imutáveis:

```
$this->response->withHeader('X-CakePHP', 'yes!');
```

No código acima, a resposta não terá o cabeçalho `X-CakePHP`, pois o valor de retorno do método `withHeader()` não foi mantido. Para corrigir o código acima, você escreveria:

```
$this->response = $this->response->withHeader('X-CakePHP', 'yes!');
```

Cookie Collections

class Cake\Http\Cookie**CookieCollection**

Os objetos `CookieCollection` são acessíveis a partir dos objetos de solicitação e resposta. Eles permitem que você interaja com grupos de cookies usando padrões imutáveis, que permitem preservar a imutabilidade da solicitação e resposta.

Criando Cookies

class Cake\Http\Cookie**Cookie**

Os objetos `Cookie` podem ser definidos por meio de objetos construtores ou usando a interface fluente que segue padrões imutáveis:

```
use Cake\Http\Cookie\Cookie;

// Todos os argumentos no construtor
$cookie = new Cookie(
    'remember_me', // nome
    1, // value
    new DateTime('+1 year'), // prazo de validade, se aplicável
    '/', // caminho, se aplicável
    'example.com', // doomínio, se aplicável
    false, // somente seguro?
    true // somente HTTP?
);

// Usando os métodos do construtor
$cookie = (new Cookie('remember_me'))
    ->withValue('1')
    ->withExpiry(new DateTime('+1 year'))
    ->withPath('/')
    ->withDomain('example.com')
```

(continues on next page)

(continuação da página anterior)

```
    ->withSecure(false)
    ->withHttpOnly(true);
```

Depois de criar um cookie, você pode adicioná-lo a um `CookieCollection` novo ou existente:

```
use Cake\Http\Cookie\CookieCollection;

// Crie uma nova coleção
$cookies = new CookieCollection([$cookie]);

// Adicionar a uma coleção existente
$cookies = $cookies->add($cookie);

// Remover um cookie pelo nome
$cookies = $cookies->remove('remember_me');
```

Nota: Lembre-se de que as coleções são imutáveis e a adição de cookies ou a remoção de cookies de uma coleção cria um *novo* objeto de coleção.

Objetos de cookie podem ser adicionados às respostas:

```
// Adiciona um cookie
$response = $this->response->withCookie($cookie);

// Substitui inteiramente uma coleção de cookie
$response = $this->response->withCookieCollection($cookies);
```

Os cookies definidos como respostas podem ser criptografados usando o *Middleware de Cookie Criptografado*.

Lendo Cookies

Depois de ter uma instância `CookieCollection`, você pode acessar os cookies que ela contém:

```
// Verifica se o cookie existe
$cookies->has('remember_me');

// Obter o número de cookies na coleção
count($cookies);

// Obter uma instância de cookie
$cookie = $cookies->get('remember_me');
```

Depois de ter um objeto `Cookie`, você pode interagir com seu estado e modificá-lo. Lembre-se de que os cookies são imutáveis, portanto, você precisará atualizar a coleção se modificar um cookie:

```
// Obtenha o valor
$value = $cookie->getValue()

// Acessar dados dentro de um valor JSON
$id = $cookie->read('User.id');
```

(continues on next page)

(continuação da página anterior)

```
// Verifica o estado
$cookie->isHttpOnly();
$cookie->isSecure();
```

CAPÍTULO 10

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 extendam. 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 sobreescrever 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 *Conectando Rotas* 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 **templates/Recipes/view.php**, **templates/Recipes/share.php** e **templates/Recipes/search.php**. 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 **templates/Recipes/search.php** será renderizado:

```
namespace App\Controller;

class RecipesController extends AppController
{
```

(continues on next page)

(continuação da página anterior)

```
// ...
    public function search()
    {
        // Render the view in templates/Recipes/search.php
        $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 templates/element/ajaxreturn.php
$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 **templates/Posts/custom_file.php** ao invés de **templates/Posts/my_action.php**

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/templates/UserDetails/custom_file.php**

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 intialize()
{
    parent::initialize();
    $this->loadComponent('Csrf');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

property Cake\Controller\Controller::$**components**

A propriedade `$components` em seus controllers permitem a você configurar components. Components configurados e suas dependências serão criados pelo CakePHP para você. Leia a seção *Configurando Componentes* para mais informações. Como mencionado anteriormente, a propriedade `$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 ausêntes.

`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 *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, *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 **templates/Pages/home.php**. Se você criar o arquivo **templates/Pages/about.php**, você poderá acessá-lo em **http://example.com/pages/about**. Fique a vontade para alterar esse controller para atender suas necessacidades ou mesmo excluí-lo.

Quando você cria sua aplicação pelo Composer, o `PagesController` vai ser criado na pasta **src/Controller/**.

Componentes

Componentes são pacotes de lógica compartilhados entre controladores. O CakePHP vem com um conjunto fantástico de componentes principais que você pode usar para ajudar em várias tarefas comuns. Você também pode criar seus próprios componentes. Se você deseja copiar e colar coisas entre controladores, considere criar seu próprio componente para conter a funcionalidade. A criação de componentes mantém o código do controlador limpo e permite reutilizar o código entre diferentes controladores.

Para mais informações sobre os componentes incluídos no CakePHP, consulte o capítulo para cada componente:

AuthComponent

class `AuthComponent`(*ComponentCollection $collection*, *array $config = []*)

Identificar, autenticar e autorizar usuários é uma parte comum de quase todos os aplicativos da web. No CakePHP, o AuthComponent fornece uma maneira conectável de executar essas tarefas. AuthComponent permite combinar objetos de autenticação e objetos de autorização para criar maneiras flexíveis de identificar e verificar a autorização do usuário.

Obsoleto desde a versão 4.0.0: O AuthComponent está obsoleto a partir da versão 4.0.0 e será substituído pelos plugins authorization[65] e authentication[66] .

Sugestão de Leitura Antes de Continuar

A configuração da autenticação requer várias etapas, incluindo a definição de uma tabela de usuários, a criação de um modelo, controlador e visualizações, etc.

Tudo isso é abordado passo a passo em *CMS Tutorial*.

Se você está procurando soluções de autenticação e/ou autorização existentes para o CakePHP, consulte a seção Authentication and Authorization[67] da lista Awesome CakePHP.

Autenticação

Autenticação é o processo de identificar usuários pelas credenciais fornecidas e garantir que os usuários sejam quem eles dizem que são. Geralmente, isso é feito através de um nome de usuário e senha, que são verificados em uma lista conhecida de usuários. No CakePHP, existem várias maneiras internas de autenticar usuários armazenados no seu aplicativo.

- `FormAuthenticate` permite autenticar usuários com base nos dados do formulário POST. Geralmente, este é um formulário de login no qual os usuários inserem informações.
- `BasicAuthenticate` permite autenticar usuários usando a autenticação HTTP básica.
- `DigestAuthenticate` permite autenticar usuários usando o Diges Autenticação HTTP.

Por padrão, `AuthComponent` usa `FormAuthenticate`.

[65] https://book.cakephp.org/authorization/
[66] https://book.cakephp.org/authentication/
[67] https://github.com/FriendsOfCake/awesome-cakephp/blob/master/README.md#authentication-and-authorization

Escolhendo um Tipo de Autenticação

Geralmente, você deseja oferecer autenticação baseada em formulário. É o mais fácil para os usuários que usam um navegador da web. Se você estiver criando uma API ou serviço da web, convém considerar a autenticação básica ou digerir a autenticação. As principais diferenças entre Digest e autenticação básica estão relacionadas principalmente à maneira como as senhas são tratadas. Na autenticação básica, o nome de usuário e a senha são transmitidos como texto sem formatação para o servidor. Isso torna a autenticação básica inadequada para aplicativos sem SSL, pois você acabaria expondo senhas confidenciais. A autenticação Digest usa um hash de resumo do nome de usuário, senha e alguns outros detalhes. Isso torna a autenticação Digest mais apropriada para aplicativos sem criptografia SSL.

Você também pode usar sistemas de autenticação como o OpenID também; no entanto, o OpenID não faz parte do núcleo do CakePHP.

Configurando Manipuladores de Autenticação

Você configura manipuladores de autenticação usando a configuração `authenticate`. Você pode configurar um ou muitos manipuladores para autenticação. O uso de vários manipuladores permite oferecer suporte a diferentes maneiras de efetuar logon nos usuários. Ao efetuar logon nos usuários, os manipuladores de autenticação são verificados na ordem em que são declarados. Quando um manipulador conseguir identificar o usuário, nenhum outro manipulador será verificado. Por outro lado, você pode interromper toda a autenticação lançando uma exceção. Você precisará capturar todas as exceções lançadas e manipulá-las conforme necessário.

Você pode configurar manipuladores de autenticação nos métodos `beforeFilter()` ou `initialize()` do seu controlador. Você pode passar informações de configuração para cada objeto de autenticação usando uma matriz:

```
// Configuração simples
$this->Auth->setConfig('authenticate', ['Form']);

// Passando as configurações
$this->Auth->setConfig('authenticate', [
    'Basic' => ['userModel' => 'Members'],
    'Form' => ['userModel' => 'Members']
]);
```

No segundo exemplo, você notará que tivemos que declarar a chave `userModel` duas vezes. Para ajudá-lo a manter seu código DRY, você pode usar a chave `all`. Essa chave especial permite definir configurações que são passadas para todos os objetos anexados. A chave `all` também é exposta como `AuthComponent::ALL`:

```
// Passando configurações usando 'all'
$this->Auth->setConfig('authenticate', [
    AuthComponent::ALL => ['userModel' => 'Members'],
    'Basic',
    'Form'
]);
```

No exemplo acima, `Form` e `Basic` obterão as configurações definido para a chave “all”. Quaisquer configurações passadas para um objeto de autenticação específico substituirão a chave correspondente na chave “all”. Os objetos de autenticação principal suportam as seguintes chaves de configuração.

- `fields` Os campos a serem usados para identificar um usuário. Você pode usar as chaves `username` e `password` para especificar seus campos de nome de usuário e senha, respectivamente.
- `userModel` O nome do modelo da tabela users; o padrão é Users.
- `finder` O método finder a ser usado para buscar um registro do usuário. O padrão é “all”.

- `passwordHasher` Classe de hasher de senha; O padrão é `Default`.

Para configurar campos diferentes para o usuário no seu método `initialize()`:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email', 'password' => 'passwd']
            ]
        ]
    ]);
}
```

Não coloque outras chaves de configuração `Auth`, como `authError`, `loginAction`, etc., dentro do elemento `authenticate` ou `Form`. Eles devem estar no mesmo nível da chave de autenticação. A configuração acima com outro exemplo de configuração para autenticação deve se parecer com:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'loginAction' => [
            'controller' => 'Users',
            'action' => 'login',
            'plugin' => 'Users'
        ],
        'authError' => 'Did you really think you are allowed to see that?',
        'authenticate' => [
            'Form' => [
                'fields' => ['username' => 'email']
            ]
        ],
        'storage' => 'Session'
    ]);
}
```

Além da configuração comum, a autenticação básica suporta as seguintes chaves:

- `realm` O domínio a ser autenticado. O padrão é `env('SERVER_NAME')`.

Além da configuração comum, a autenticação Digest suporta as seguintes chaves:

- `realm` Para autenticação de domínio. O padrão é o nome do servidor.
- `nonce` Um nonce usado para autenticação. O padrão é `uniqid()`.
- `qop` O padrão é auth; nenhum outro valor é suportado no momento.
- `opaque` Uma sequência que deve ser retornada inalterada pelos clientes. O padrão é `md5($config['realm'])`.

Nota: Para encontrar o registro do usuário, o banco de dados é consultado apenas usando o nome de usuário. A verificação da senha é feita em PHP. Isso é necessário porque algoritmos de hash como bcrypt (que é usado por padrão)

geram um novo hash a cada vez, mesmo para a mesma string e você não pode simplesmente fazer uma comparação simples de strings no SQL para verificar se a senha corresponde.

Personalizando a Consulta de Localização

Você pode personalizar a consulta usada para buscar o registro do usuário usando a opção `finder` na opção de autenticação da classe:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'finder' => 'auth'
            ]
        ],
    ]);
}
```

Isso exigirá que seu `UsersTable` tenha o método localizador `findAuth()`. No exemplo mostrado abaixo, a consulta é modificada para buscar apenas os campos obrigatórios e adicionar uma condição. Você deve garantir que você selecione os campos necessários para autenticar um usuário, como `username` e `password`:

```
public function findAuth(\Cake\ORM\Query $query, array $options)
{
    $query
        ->select(['id', 'username', 'password'])
        ->where(['Users.active' => 1]);

    return $query;
}
```

Identificando Usuários e Efetuando Login

`AuthComponent::identify()`

Você precisa chamar manualmente `$this->Auth->identity()` para identificar o usuário usando as credenciais fornecidas na solicitação. Em seguida, use `$this->Auth->setUser()` para conectar o usuário, ou seja, salve as informações do usuário na sessão.

Ao autenticar usuários, os objetos de autenticação anexados são verificados na ordem em que estão. Depois que um dos objetos pode identificar o usuário, nenhum outro objeto é verificado. Uma função de login como exemplo para trabalhar com um formulário de login pode se parecer com:

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
```

(continues on next page)

(continuação da página anterior)

```
            return $this->redirect($this->Auth->redirectUrl());
        } else {
            $this->Flash->error(__('Username or password is incorrect'));
        }
    }
}
```

O código acima tentará primeiro identificar um usuário usando os dados do POST. Se for bem-sucedido, definimos as informações do usuário para a sessão, para que elas persistam nas solicitações e, em seguida, redirecionamos para a última página que eles estavam visitando ou para uma URL especificada na configuração `loginRedirect`. Se o logon não for bem-sucedido, uma mensagem flash será definida.

Aviso: `$this->Auth->setUser($data)` registrará o usuário com todos os dados passados para o método. Na verdade, ele não verifica as credenciais em uma classe de autenticação.

Redirecionando Usuários após o Login

`AuthComponent::redirectUrl()`

Depois de fazer o login de um usuário, você geralmente desejará redirecioná-lo de volta para onde eles vieram. Passe um URL para definir o destino ao qual um usuário deve ser redirecionado após o login.

Se nenhum parâmetro for passado, a URL retornada usará as seguintes regras:

- Retorna a URL normalizada do valor da string de consulta `redirect`, se estiver presente e no mesmo domínio em que o aplicativo atual estiver sendo executado.
- Se não houver um valor de string/sessão de consulta e houver uma configuração com `loginRedirect`, o valor `loginRedirect` será retornado.
- Se não houver valor de redirecionamento e nenhum `loginRedirect`, `/` será retornado.

Criando Sistemas de Autenticação sem Estado

Basic e Digest são esquemas de autenticação sem estado e não requerem um POST ou um formulário inicial. Se você estiver usando apenas autenticadores basic/digest, não precisará de uma ação de login no seu controlador. A autenticação sem estado verificará novamente as credenciais do usuário em cada solicitação, isso cria uma pequena quantidade de sobrecarga adicional, mas permite que os clientes efetuem login sem usar cookies e torna o AuthComponent mais adequado para a criação de APIs.

Para autenticadores sem estado, a configuração `` storage`` deve ser definida como `Memory` para que o AuthComponent não use uma sessão para armazenar o registro do usuário. Você também pode querer configurar config `unauthorizedRedirect` para `false`, para que AuthComponent gere uma `ForbiddenException` em vez do comportamento padrão de redirecionar para o referenciador.

A opção `unauthorizedRedirect` se aplica apenas a usuários autenticados. Quando um usuário ainda não está autenticado e você não deseja que ele seja redirecionado, será necessário carregar um ou mais autenticadores sem estado, como `Basic` ou `Digest`.

Objetos de autenticação podem implementar um método `getUser()` que pode ser usado para oferecer suporte a sistemas de login de usuário que não dependem de cookies. Um método getUser típico examina a solicitação/ambiente

e usa as informações para confirmar a identidade do usuário. A autenticação HTTP Basic, por exemplo, usa `$_SERVER['PHP_AUTH_USER']` e `$_SERVER['PHP_AUTH_PW']` para os campos de nome de usuário e senha.

Nota: Caso a autenticação não funcione como o esperado, verifique se as consultas são executadas (consulte `BaseAuthenticate::_query($username)`). Caso nenhuma consulta seja executada, verifique se `$_SERVER['PHP_AUTH_USER']` e `$_SERVER['PHP_AUTH_PW']` são preenchidos pelo servidor web. Se você estiver usando o Apache com FastCGI-PHP, poderá ser necessário adicionar esta linha ao seu arquivo **.htaccess** no webroot:

```
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization},L]
```

Em cada solicitação, esses valores, `PHP_AUTH_USER` e `PHP_AUTH_PW`, são usados para identificar novamente o usuário e garantir que ele seja o usuário válido. Assim como no método `authenticate()` do objeto de autenticação, o método `getUser()` deve retornar uma matriz de informações do usuário sobre o sucesso ou `false` em caso de falha.

```
public function getUser(ServerRequest $request)
{
    $username = env('PHP_AUTH_USER');
    $pass = env('PHP_AUTH_PW');

    if (empty($username) || empty($pass)) {
        return false;
    }

    return $this->_findUser($username, $pass);
}
```

A seguir, é apresentado como você pode implementar o método getUser para autenticação HTTP básica. O método `_findUser()` faz parte de `BaseAuthenticate` e identifica um usuário com base em um nome de usuário e senha.

Usando Autenticação Básica

A autenticação básica permite criar uma autenticação sem estado que pode ser usada em aplicativos de intranet ou em cenários simples da API. As credenciais de autenticação básica serão verificadas novamente em cada solicitação.

> **Aviso:** A autenticação básica transmite credenciais em texto sem formatação. Você deve usar HTTPS ao usar a autenticação básica.

Para usar a autenticação básica, você precisará configurar o AuthComponent:

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Basic' => [
            'fields' => ['username' => 'username', 'password' => 'api_key'],
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Aqui, usamos o nome de usuário + chave da API como nossos campos e usamos o modelo Usuários.

Criando Chaves de API para Autenticação Básica

Como o HTTP básico envia credenciais em texto sem formatação, não é aconselhável que os usuários enviem sua senha de login. Em vez disso, geralmente é usada uma chave de API. Você pode gerar esses tokens de API aleatoriamente usando bibliotecas do CakePHP:

```
namespace App\Model\Table;

use Cake\Auth\DefaultPasswordHasher;
use Cake\Utility\Text;
use Cake\Event\EventInterface;
use Cake\ORM\Table;
use Cake\Utility\Security;

class UsersTable extends Table
{
    public function beforeSave(EventInterface $event)
    {
        $entity = $event->getData('entity');

        if ($entity->isNew()) {
            $hasher = new DefaultPasswordHasher();

            // Gera uma API 'token'
            $entity->api_key_plain = Security::hash(Security::randomBytes(32), 'sha256',␣
↪false);

            // Criptografe o token para que BasicAuthenticate
            // possa verificá-lo durante o login.
            $entity->api_key = $hasher->hash($entity->api_key_plain);
        }

        return true;
    }
}
```

O exemplo acima gera um hash aleatório para cada usuário conforme eles são salvos. O código acima assume que você tem duas colunas `api_key` - para armazenar a chave da API hash e `api_key_plain` - para a versão em texto sem formatação da chave da API, para que possamos exibi-la ao usuário posteriormente. Usar uma chave em vez de uma senha significa que, mesmo em HTTP simples, seus usuários podem usar um token simples em vez da senha original. Também é aconselhável incluir lógica que permita que as chaves da API sejam regeneradas a pedido de um usuário.

Usando Autenticação Digest

A autenticação Digest oferece um modelo de segurança aprimorado em relação à autenticação básica, pois as credenciais do usuário nunca são enviadas no cabeçalho da solicitação. Em vez disso, um hash é enviado.

Para usar a autenticação Digest, você precisará configurar o `AuthComponent`:

```
$this->loadComponent('Auth', [
    'authenticate' => [
        'Digest' => [
            'fields' => ['username' => 'username', 'password' => 'digest_hash'],
```

(continues on next page)

(continuação da página anterior)

```
            'userModel' => 'Users'
        ],
    ],
    'storage' => 'Memory',
    'unauthorizedRedirect' => false
]);
```

Aqui, estamos usando o nome de usuário + digest_hash como nossos campos e também usamos o modelo Users.

Hashing de Senhas para Autenticação Digest

Como a autenticação Digest requer um hash de senha no formato definido pelo RFC, para hash corretamente uma senha para uso com a autenticação Digest, você deve usar a função de hash de senha especial em `DigestAuthenticate`. Se você combinar a autenticação digest com outras estratégias de autenticação, também é recomendável que você armazene a senha digest em uma coluna separada, a partir do hash da senha normal:

```
namespace App\Model\Table;

use Cake\Auth\DigestAuthenticate;
use Cake\Event\EventInterface;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function beforeSave(EventInterface $event)
    {
        $entity = $event->getData('entity');

        // Fazendo a senha para autenticação digest
        $entity->digest_hash = DigestAuthenticate::password(
            $entity->username,
            $entity->plain_password,
            env('SERVER_NAME')
        );

        return true;
    }
}
```

As senhas para autenticação Digest precisam de um pouco mais de informações do que outros hashes de senha, com base no RFC para autenticação Digest.

Nota: O terceiro parâmetro de `DigestAuthenticate::password()` deve corresponder ao valor de configuração "realm" definido quando DigestAuthentication foi configurado em `AuthComponent::$authenticate`. O padrão é `env('SCRIPT_NAME')`. Você pode usar uma string estática se desejar hashes consistentes em vários ambientes.

Criando Objetos de Autenticação Personalizados

Como os objetos de autenticação são conectáveis, você pode criar objetos de autenticação personalizados em seu aplicativo ou plug-in. Se, por exemplo, você desejasse criar um objeto de autenticação OpenID. Em **src/Auth/OpenidAuthenticate.php**, você pode colocar o seguinte:

```
namespace App\Auth;

use Cake\Auth\BaseAuthenticate;
use Cake\Http\ServerRequest;
use Cake\Http\Response;

class OpenidAuthenticate extends BaseAuthenticate
{
    public function authenticate(ServerRequest $request, Response $response)
    {
        // Faça coisas para o OpenID aqui.
        // Retorne uma matriz do usuário se eles puderem autenticar o usuário,
        // retorne false se não.
    }
}
```

Os objetos de autenticação devem retornar `false` se não puderem identificar o usuário e uma matriz de informações do usuário, se puderem. Não é necessário que você estenda `BaseAuthenticate`, apenas que seu objeto de autenticação implemente `Cake\Event\EventListenerInterface`. A classe `BaseAuthenticate` fornece vários métodos úteis que são comumente usados. Você também pode implementar um método `getUser()` se o seu objeto de autenticação precisar suportar autenticação sem estado ou sem cookie. Consulte as seções sobre autenticação básica e digest abaixo para obter mais informações.

`AuthComponent` dispara dois eventos, `Auth.afterIdentify` e `Auth.logout`, depois que um usuário é identificado e antes que o usuário seja desconectado, respectivamente. Você pode definir funções de retorno de chamada para esses eventos retornando uma matriz de mapeamento do método `managedEvents()` da sua classe de autenticação:

```
public function implementedEvents()
{
    return [
        'Auth.afterIdentify' => 'afterIdentify',
        'Auth.logout' => 'logout'
    ];
}
```

Usando Objetos de Autenticação Personalizados

Depois de criar seus objetos de autenticação personalizados, você pode usá-los incluindo-os na matriz de autenticação do `AuthComponent`:

```
$this->Auth->setConfig('authenticate', [
    'Openid', // app authentication object.
    'AuthBag.Openid', // plugin authentication object.
]);
```

Nota: Observe que, ao usar notação simples, não há palavra "Authenticate" ao iniciar o objeto de autenticação. Em

vez disso, se você estiver usando namespace, precisará definir o namespace completo da classe, incluindo a palavra "Authenticate".

Manipulando Solicitações Não Autenticadas

Quando um usuário não autenticado tenta acessar uma página protegida primeiro, o método `unauthenticated()` do último autenticador da cadeia é chamado. O objeto de autenticação pode lidar com o envio de resposta ou redirecionamento retornando um objeto de resposta para indicar que nenhuma ação adicional é necessária. Devido a isso, é importante a ordem na qual você especifica o provedor de autenticação na configuração `authenticate`.

Se o autenticador retornar nulo, `AuthComponent` redirecionará o usuário para a ação de login. Se for uma solicitação AJAX e a configuração `ajaxLogin` for especificada, esse elemento será renderizado, caso contrário, um código de status HTTP 403 será retornado.

Exibindo Mensagens Flash Relacionadas à Autenticação

Para exibir as mensagens de erro da sessão que o Auth gera, você precisa adicionar o seguinte código ao seu layout. Adicione as duas linhas a seguir ao arquivo **templates/layout/default.php** na seção body:

```
echo $this->Flash->render();
```

Você pode personalizar as mensagens de erro e as configurações do flash que o `AuthComponent` usa. Usando a configuração `flash`, você pode configurar os parâmetros que o `AuthComponent` usa para definir mensagens em flash. As chaves disponíveis são

- `key` - A chave a ser usada é padronizada como "default".
- `element` - O nome do elemento a ser usado para renderização, o padrão é null.
- `params` - A matriz de parâmetros adicionais a serem usados, o padrão é `[]`.

Além das configurações de mensagens flash, você pode personalizar outras mensagens de erro que o `AuthComponent` usa. Nas configurações `beforeFilter()` do seu controlador ou componente, você pode usar `authError` para personalizar o erro usado quando a autorização falha:

```
$this->Auth->setConfig('authError', "Woopsie, you are not authorized to access this area.
↪");
```

Às vezes, você deseja exibir o erro de autorização somente após o usuário já estar conectado. Você pode suprimir esta mensagem definindo seu valor como booleano `false`.

Nas configurações `beforeFilter()` ou no componente do seu controlador:

```
if (!$this->Auth->user()) {
    $this->Auth->setConfig('authError', false);
}
```

Hashing de Senhas

Você é responsável por fazer o hash das senhas antes que elas persistam no banco de dados, a maneira mais fácil é usar uma função setter na sua entidade User:

```
namespace App\Model\Entity;

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

class User extends Entity
{

    // ...

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

    // ...
}
```

`AuthComponent` é configurado por padrão para usar o `DefaultPasswordHasher` ao validar credenciais do usuário, portanto, nenhuma configuração adicional é necessária para autenticar usuários.

`AuthComponent` é configurado como padrão para usar o `DefaultPasswordHasher` para validar credenciais do usuário, portanto, nenhuma configuração adicional é necessária para autenticação de usuários.

O `DefaultPasswordHasher` usa o algoritmo de hash bcrypt internamente, que é uma das soluções mais fortes de hash de senha usadas no setor. Embora seja recomendável usar essa classe de hasher de senha, pode ser que você esteja gerenciando um banco de dados de usuários cuja senha foi usada um tipo de hash diferente.

Criando Classes Personalizadas de Hasher de Senha

Para usar um hasher de senha diferente, você precisa criar a classe em **src/Auth/LegacyPasswordHasher.php** e implementar os métodos `hash()` e `check()`. Esta classe precisa estender a classe `AbstractPasswordHasher`:

```
namespace App\Auth;

use Cake\Auth\AbstractPasswordHasher;

class LegacyPasswordHasher extends AbstractPasswordHasher
{

    public function hash($password)
    {
        return sha1($password);
    }

    public function check($password, $hashedPassword)
```

(continues on next page)

(continuação da página anterior)

```
    {
        return sha1($password) === $hashedPassword;
    }
}
```

Em seguida, você deve configurar o `AuthComponent` para usar o seu hasher de senha customizado:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Legacy',
                ]
            ]
        ]
    ]);
}
```

Oferecer suporte a sistemas legados é uma boa idéia, mas é ainda melhor manter seu banco de dados com os mais recentes avanços de segurança. A seção a seguir explica como migrar de um algoritmo de hash para o padrão do CakePHP.

Alterando Algoritmos de Hash

O CakePHP fornece uma maneira limpa de migrar as senhas de seus usuários de um algoritmo para outro, isso é alcançado através da classe `FallbackPasswordHasher`. Supondo que você esteja migrando seu aplicativo do CakePHP 2.x que usa hashes de senha `sha1`, você pode configurar o `AuthComponent` da seguinte forma:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Auth', [
        'authenticate' => [
            'Form' => [
                'passwordHasher' => [
                    'className' => 'Fallback',
                    'hashers' => [
                        'Default',
                        'Weak' => ['hashType' => 'sha1']
                    ]
                ]
            ]
        ]
    ]);
}
```

O primeiro nome que aparece na chave `hashers` indica qual das classes é a preferida, mas retornará para as outras na lista se a verificação não tiver êxito.

Ao usar o `WeakPasswordHasher`, você precisará definir o `Security.salt` para configurar o valor para garantir que as senhas sejam transformadas.

Para atualizar as senhas de usuários antigos rapidamente, você pode alterar a função de login de acordo:

```
public function login()
{
    if ($this->request->is('post')) {
        $user = $this->Auth->identify();
        if ($user) {
            $this->Auth->setUser($user);
            if ($this->Auth->authenticationProvider()->needsPasswordRehash()) {
                $user = $this->Users->get($this->Auth->user('id'));
                $user->password = $this->request->getData('password');
                $this->Users->save($user);
            }

            return $this->redirect($this->Auth->redirectUrl());
        }
        ...
    }
}
```

Como você pode ver, estamos apenas definindo a senha simples novamente, para que a função setter na entidade faça hash na senha, como mostrado no exemplo anterior, e salve a entidade.

Logon Manual de Usuários

`AuthComponent::setUser`(*array $user*)

Às vezes, surge a necessidade de fazer o login manual de um usuário, como logo após ele se registrar no seu aplicativo. Você pode fazer isso chamando `$this->Auth->setUser()` com os dados do usuário que deseja "logar":

```
public function register()
{
    $user = $this->Users->newEntity($this->request->getData());
    if ($this->Users->save($user)) {
        $this->Auth->setUser($user->toArray());

        return $this->redirect([
            'controller' => 'Users',
            'action' => 'home'
        ]);
    }
}
```

Aviso: Certifique-se de adicionar manualmente o novo ID do usuário à matriz passada para o método `setUser()`. Caso contrário, você não terá o ID do usuário disponível.

Acessando o Usuário Conectado

AuthComponent::**user**(*$key = null*)

Depois que um usuário está logado, muitas vezes você precisará de algumas informações específicas sobre o usuário atual. Você pode acessar o usuário conectado no momento usando `AuthComponent::user()`:

```
// De dentro de um controlador ou outro componente.
$this->Auth->user('id');
```

Se o usuário atual não estiver conectado ou a chave não existir, o valor nulo será retornado.

Logout de Usuários

AuthComponent::**logout**()

Eventualmente, você desejará uma maneira rápida de autenticar alguém e redirecioná-lo para onde ele precisa ir. Esse método também é útil se você deseja fornecer um link "Desconectar-se" dentro da área de um membro do seu aplicativo:

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

É difícil realizar logoff de usuários que efetuaram logon com autenticação Digest ou Basic para todos os clientes. A maioria dos navegadores retém credenciais pelo período em que ainda estão abertos. Alguns clientes podem ser forçados a sair, enviando um código de status 401. Alterar o domínio de autenticação é outra solução que funciona para alguns clientes.

Decidindo Quando Executar a Autenticação

Em alguns casos, você pode querer usar `$this->Auth->user()` no método `beforeFilter()`. Isso é possível usando a chave de configuração `checkAuthIn`. As alterações a seguir, para o qual as verificações de autenticação inicial devem ser feitas:

```
// Configure AuthComponent para autenticar em initialize ()
$this->Auth->setConfig('checkAuthIn', 'Controller.initialize');
```

O valor padrão para `checkAuthIn` é `'Controller.startup'`, mas usando a autenticação inicial `'Controller.initialize'` é feita antes do método `beforeFilter()`.

Autorização

Autorização é o processo de garantir que um usuário identificado/autenticado tenha permissão para acessar os recursos que está solicitando. Se ativado, o `AuthComponent` pode verificar automaticamente os manipuladores de autorização e garantir que os usuários conectados tenham permissão para acessar os recursos que estão solicitando. Existem vários manipuladores de autorização internos e você pode criar personalizações para o seu aplicativo ou como parte de um plug-in.

- `ControllerAuthorize` Chama `isAuthorized()` no controlador ativo e usa o retorno para autorizar um usuário. Geralmente, é a maneira mais simples de autorizar usuários.

Nota: O adaptador `ActionsAuthorize` e `CrudAuthorize` disponível no CakePHP 2.x foram agora movidos para um plugin separado cakephp/acl[68].

Configurando Manipuladores de Autorização

Você configura manipuladores de autorização usando a chave de configuração `authorize`. Você pode configurar um ou muitos manipuladores para autorização. O uso de vários manipuladores permite oferecer suporte a diferentes maneiras de verificar a autorização. Quando os manipuladores de autorização são verificados, eles serão chamados na ordem em que são declarados. Os manipuladores devem retornar `false`, se não conseguirem verificar a autorização ou se a verificação falhar. Os manipuladores devem retornar `true` se puderem verificar a autorização com êxito. Os manipuladores serão chamados em sequência até que um passe. Se todas as verificações falharem, o usuário será redirecionado para a página de onde veio. Além disso, você pode interromper toda a autorização lançando uma exceção. Você precisará capturar todas as exceções lançadas e lidar com elas.

Você pode configurar manipuladores de autorização nos métodos `beforeFilter()` ou `initialize()` do seu controlador. Você pode passar informações de configuração para cada objeto de autorização, usando uma matriz:

```
// Configuração básica
$this->Auth->setConfig('authorize', ['Controller']);

// Passando configurações
$this->Auth->setConfig('authorize', [
    'Actions' => ['actionPath' => 'controllers/'],
    'Controller'
]);
```

Assim como `authenticate`, `authorize`, ajuda a manter seu código DRY, usando a chave `all`. Essa chave especial permite definir configurações que são passadas para todos os objetos anexados. A chave `all` também é exposta como `AuthComponent::ALL`:

```
// Passando as configurações usando 'all'
$this->Auth->setConfig('authorize', [
    AuthComponent::ALL => ['actionPath' => 'controllers/'],
    'Actions',
    'Controller'
]);
```

No exemplo acima, as ações `Actions` e `` Controller`` receberão as configurações definidas para a chave “all”. Quaisquer configurações passadas para um objeto de autorização específico substituirão a chave correspondente na chave “all”.

Se um usuário autenticado tentar acessar uma URL que ele não está autorizado a acessar, ele será redirecionado de volta ao referenciador. Se você não desejar esse redirecionamento (principalmente necessário ao usar o adaptador de autenticação sem estado), defina a opção de configuração `unauthorizedRedirect` para `false`. Isso faz com que o `AuthComponent` gere uma `ForbiddenException` em vez de redirecionar.

[68] https://github.com/cakephp/acl

Criando Objetos de Autorização Personalizados

Como os objetos de autorização são conectáveis, você pode criar objetos de autorização personalizados em seu aplicativo ou plug-in. Se, por exemplo, você desejasse criar um objeto de autorização LDAP. Em **src/Auth/LdapAuthorize.php**, você pode colocar o seguinte:

```
namespace App\Auth;

use Cake\Auth\BaseAuthorize;
use Cake\Http\ServerRequest;

class LdapAuthorize extends BaseAuthorize
{
    public function authorize($user, ServerRequest $request)
    {
        // Faça coisas para o LDAP aqui.
    }
}
```

Os objetos de autorização devem retornar `false` se o acesso do usuário for negado ou se o objeto não puder executar uma verificação. Se o objeto puder verificar o acesso do usuário, `true` deve ser retornado. Não é necessário que você estenda `BaseAuthorize`, apenas que seu objeto de autorização implemente um método `authorize()`. A classe `BaseAuthorize` fornece vários métodos úteis que são comumente usados.

Usando Objetos de Autorização Personalizados

Depois de criar seu objeto de autorização personalizado, você pode usá-lo incluindo-o na matriz de autorização do `AuthComponent`:

```
$this->Auth->setConfig('authorize', [
    'Ldap', // objeto de autorização do aplicativo.
    'AuthBag.Combo', // plugin autoriza objeto.
]);
```

Usando Sem Autorização

Se você não quiser usar nenhum dos objetos de autorização internos e quiser lidar com coisas totalmente fora de `AuthComponent`, poderá definir `$this->Auth->setConfig('authorize', false);`. Por padrão, `AuthComponent` começa com `authorize` definido como `false`. Se você não usar um esquema de autorização, verifique você mesmo a autorização no `beforeFilter()` do seu controlador ou com outro componente.

Tornando Métodos Públicos

`AuthComponent::allow`(*$actions = null*)

Muitas vezes, há ações do controlador que você deseja manter totalmente públicas ou que não exigem que os usuários façam login. `AuthComponent` é pessimista no padrão para negar acesso. Você pode marcar métodos como métodos públicos usando `AuthComponent::allow()`. Ao marcar ações como públicas, o `AuthComponent` não procurará um usuário conectado nem autorizará a verificação de objetos:

```
// Permitir todas as ações
$this->Auth->allow();

// Permitir apenas a ação index.
$this->Auth->allow('index');

// Permitir apenas as ações de view e index.
$this->Auth->allow(['view', 'index']);
```

Ao chamá-lo de vazio, você permite que todas as ações sejam públicas. Para uma única ação, você pode fornecer o nome da ação como uma sequência. Caso contrário, use uma matriz.

Nota: Você não deve adicionar a ação «login» do seu `UsersController` na lista de permissões. Fazer isso causaria problemas com o funcionamento normal de `AuthComponent`.

Fazendo Métodos Exigir Autorização

`AuthComponent::deny`(*$actions = null*)

Por padrão, todas as ações requerem autorização. No entanto, depois de tornar os métodos públicos, você deseje revogar o acesso público. Você pode fazer isso usando `AuthComponent::deny()`:

```
// Negar todas as ações.
$this->Auth->deny();

// Negar uma ação
$this->Auth->deny('add');

// Nega um grupo de ações.
$this->Auth->deny(['add', 'edit']);
```

Ao chamá-lo de vazio, você nega todas as ações. Para um único método, você pode fornecer o nome da ação como uma sequência. Caso contrário, use uma matriz.

Usando ControllerAuthorize

ControllerAuthorize permite manipular verificações de autorização em um retorno de chamada do controlador. Isso é ideal quando você possui uma autorização muito simples ou precisa usar uma combinação de modelos e componentes para fazer sua autorização e não deseja criar um objeto de autorização personalizado.

O retorno de chamada é sempre chamado de `isAuthorized()` e deve retornar um valor booleano para permitir ou não ao usuário acessar recursos na solicitação. O retorno de chamada é passado ao usuário ativo para que possa ser verificado:

```
class AppController extends Controller
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth', [
```

(continues on next page)

(continuação da página anterior)

```
            'authorize' => 'Controller',
        ]);
    }

    public function isAuthorized($user = null)
    {
        // Qualquer usuário registrado pode acessar funções públicas
        if (!$this->request->getParam('prefix')) {
            return true;
        }

        // Somente administradores podem acessar funções administrativas
        if ($this->request->getParam('prefix') === 'admin') {
            return (bool)($user['role'] === 'admin');
        }

        // Negação padrão
        return false;
    }
}
```

O retorno de chamada acima forneceria um sistema de autorização muito simples, no qual apenas usuários com role = admin poderiam acessar ações que estavam no prefixo do administrador.

Opções de configuração

Todas as configurações a seguir podem ser definidas no método `initialize()` do seu controlador ou usando `$this->Auth->setConfig()` no seu `beforeFilter()`:

ajaxLogin
: O nome de um elemento de exibição opcional a ser renderizado quando uma solicitação AJAX é feita com uma sessão inválida ou expirada.

allowedActions
: Ações do controlador para as quais a validação do usuário não é necessária.

authenticate
: Defina como uma matriz de objetos de autenticação que você deseja usar ao fazer logon de usuários. Existem vários objetos de autenticação principais; veja a seção *Sugestão de Leitura Antes de Continuar*.

authError
: Erro para exibir quando o usuário tenta acessar um objeto ou ação ao qual não tem acesso.

  Você pode impedir que a mensagem authError seja exibida definindo esse valor como booleano `false`.

authorize
: Defina como uma matriz de objetos de Autorização que você deseja usar ao autorizar usuários em cada solicitação; veja a seção *Autorização*.

flash
: Configurações a serem usadas quando o Auth precisar enviar uma mensagem flash com `FlashComponent::set()`.

  As chaves disponíveis são: - `element` - O elemento a ser usado; o padrão é "default". - `key` - A chave para usar; o padrão é "auth". - `params` - A matriz de parâmetros adicionais a serem usados; o padrão é "[]".

loginAction
Uma URL (definida como uma sequência ou matriz) para a ação do controlador que lida com logins. O padrão é `/users/login`.

loginRedirect
A URL (definida como uma sequência ou matriz) para os usuários da ação do controlador deve ser redirecionada após o login. Esse valor será ignorado se o usuário tiver um valor `Auth.redirect` em sua sessão.

logoutRedirect
A ação padrão a ser redirecionada após o logout do usuário. Enquanto `AuthComponent` não lida com o redirecionamento pós-logout, uma URL de redirecionamento será retornada de *AuthComponent::logout()*. O padrão é `loginAction`.

unauthorizedRedirect
Controla a manipulação do acesso não autorizado. Por padrão, o usuário não autorizado é redirecionado para o URL do referenciador `loginAction` ou "/". Se definido como `false`, uma exceção ForbiddenException é lançada em vez de redirecionar.

storage
Classe de armazenamento a ser usada para o registro persistente do usuário. Ao usar o autenticador sem estado, defina-o como `Memory`. O padrão é `Session`. Você pode passar as opções de configuração para a classe de armazenamento usando o formato de matriz. Por ex. Para usar uma chave de sessão personalizada, você pode definir `storage` como `['className' => 'Session', 'key' => 'Auth.Admin']`.

checkAuthIn
Nome do evento no qual as verificações de autenticação iniciais devem ser feitas. O padrão é `Controller.startup`. Você pode configurá-lo para `Controller.initialize` se você quiser que a verificação seja feita antes que o método `beforeFilter()` do controlador seja executado.

Você pode obter os valores atuais da configuração chamando `$this->Auth->getConfig()`:: apenas a opção de configuração:

```
$this->Auth->getConfig('loginAction');

$this->redirect($this->Auth->getConfig('loginAction'));
```

Isso é útil se você deseja redirecionar um usuário para a rota `login`, por exemplo. Sem um parâmetro, a configuração completa será retornada.

Testando Ações Protegidas por AuthComponent

Veja a seção *Testando Ações que Exigem Autenticação* para obter dicas sobre como testar ações do controlador protegidas por `AuthComponent`.

Flash

class Cake\Controller\Component**FlashComponent**(*ComponentCollection $collection*, *array $config = []*)

O FlashComponent fornece uma maneira de definir as mensagens de notificação únicas a serem exibidas após o processamento de um formulário ou o reconhecimento de dados. CakePHP refere-se a essas mensagens como «mensagens flash». O FlashComponent grava mensagens flash em `$_SESSION`, para serem renderizadas em uma View usando *FlashHelper*.

Configurando Mensagens em Flash

O FlashComponent fornece duas maneiras de definir mensagens em flash: seu método mágico `__call()` e seu método `set()`. Para fornecer detalhes ao seu aplicativo, o método mágico `__call()` do FlashComponent permite que você use um nome de método que mapeie para um elemento localizado no diretório **templates/element/flash**. Por convenção, os métodos camelcased serão mapeados para o nome do elemento em minúsculas e sublinhado:

```
// Usa templates/element/flash/success.php
$this->Flash->success('This was successful');

// Usa templates/element/flash/great_success.php
$this->Flash->greatSuccess('This was greatly successful');
```

Como alternativa, para definir uma mensagem de texto sem processar um elemento, você pode usar o método `set()`:

```
$this->Flash->set('This is a message');
```

Mensagens em Flash são anexadas a uma matriz internamente. Chamadas sucessivas para `set()` ou `__call()` com a mesma chave anexarão as mensagens em `$_SESSION`. Se você deseja sobrescrever as mensagens existentes ao definir uma mensagem flash, defina a opção `clear` como `true` ao configurar o componente.

Os métodos `__call()` e `set()` do FlashComponent recebem opcionalmente um segundo parâmetro, uma matriz de opções:

- `key` O padrão é "flash". A chave da matriz encontrada sob a chave `Flash` na sessão.
- `element` O padrão é `null`, mas será automaticamente definido ao usar o método mágico `__call()`. O nome do elemento a ser usado para renderização.
- `params` Uma matriz opcional de chaves/valores para disponibilizar como variáveis dentro de um elemento.
- `clear` espera um `bool` e permite excluir todas as mensagens da pilha atual e iniciar uma nova.

Um exemplo de uso dessas opções:

```
// Em seu Controller
$this->Flash->success('The user has been saved', [
    'key' => 'positive',
    'clear' => true,
    'params' => [
        'name' => $user->name,
        'email' => $user->email
    ]
]);

// Em sua View
<?= $this->Flash->render('positive') ?>

<!-- Em templates/element/flash/success.php -->
<div id="flash-<?= h($key) ?>" class="message-info success">
    <?= h($message) ?>: <?= h($params['name']) ?>, <?= h($params['email']) ?>.
</div>
```

Observe que o parâmetro `element` sempre será substituído ao usar

Observe que o parâmetro `element` sempre será substituído ao usar `__call()`. Para recuperar um elemento específico de um plugin, você deve definir o parâmetro `plugin`. Por exemplo:

```
// Em seu Controller
$this->Flash->warning('My message', ['plugin' => 'PluginName']);
```

O código acima usará o elemento **warning.php** em **plugins/PluginName/templates/element/flash** para renderizar a mensagem flash.

Nota: Por padrão, o CakePHP escapa o conteúdo das mensagens em flash para evitar scripts entre sites. Os dados do usuário em suas mensagens flash serão codificados em HTML e seguros para serem impressos. Se você deseja incluir HTML em suas mensagens em flash, é necessário passar a opção `escape` e ajustar seus modelos de mensagens em flash para permitir desativar a fuga quando a opção de escape é aprovada.

HTML Em Mensagens Flash

É possível gerar HTML em mensagens flash usando a chave de opção `'escape'`:

```
$this->Flash->info(sprintf('<b>%s</b> %s', h($highlight), h($message)), ['escape' =>␣
↪false]);
```

Certifique-se de escapar da entrada manualmente, então. No exemplo acima, `$highlights` e `$message` são entradas não HTML e, portanto, escapam.

Para obter mais informações sobre como renderizar suas mensagens em flash, consulte a seção *FlashHelper*.

Security (Segurança)

`class SecurityComponent`(*ComponentCollection $collection*, *array $config = []*)

O componente de segurança cria uma maneira fácil de integrar maior segurança ao seu aplicativo. Ele fornece métodos para várias tarefas, como:

- Restringindo quais métodos HTTP seu aplicativo aceita.
- Proteção contra violação de formulário
- Exigindo que o SSL seja usado.
- Limitar a comunicação entre controladores.

Como todos os componentes, ele é ajustado através de vários parâmetros configuráveis. Todas essas propriedades podem ser definidas diretamente ou através de métodos setter com o mesmo nome no `beforeFilter()` do seu controlador.

Ao usar o componente de segurança, você obtém automaticamente proteção contra violação de formulários. Os campos de token oculto são inseridos automaticamente nos formulários e verificados pelo componente Security.

Se você estiver usando os recursos de proteção de formulário do componente Security e outros componentes que processam dados do formulário em seus retornos de chamada `startup()`, certifique-se de colocar o Componente de Segurança antes desses componentes no método `initialize()`.

Nota: Ao usar o componente de segurança, você deve usar o FormHelper para criar seus formulários. Além disso, você não deve substituir nenhum dos atributos «name» dos campos. O componente de segurança procura determinados indicadores criados e gerenciados pelo FormHelper (especialmente aqueles criados em `create()` e `end()`). Alterar dinamicamente os campos que são enviados em uma solicitação POST (por exemplo, desativar, excluir ou criar novos campos via JavaScript) provavelmente fará com que a solicitação seja enviada como retorno a um blackhole.

Você sempre deve verificar o método HTTP que está sendo usado antes da execução para evitar efeitos colaterais. Você deve *check the HTTP method* ou usar `Cake\Http\ServerRequest::allowMethod()` para garantir que o método HTTP correto seja usado.

Como Lidar com Retornos de Chamada Blackhole

`SecurityComponent::blackHole`(*Controller $controller*, *string $error = ''*, *?SecurityException $exception = null*)

Se uma ação for restringida pelo Componente de Segurança, ela será "ocultada em preto" como uma solicitação inválida que resultará em um erro 400 por padrão. Você pode configurar esse comportamento definindo a opção de configuração `blackHoleCallback` como uma função de retorno de chamada no controlador.

Ao configurar um método de retorno de chamada, você pode personalizar como o processo do blackhole funciona:

```
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);

    $this->Security->setConfig('blackHoleCallback', 'blackhole');
}

public function blackhole($type, SecurityException $exception)
{
    if ($exception->getMessage() === 'Request is not SSL and the action is required to␣
↪be secure') {
        // Reescreva a mensagem de exceção com uma sequência traduzível.
        $exception->setMessage(__('Please access the requested page through HTTPS'));
    }

    // Lance novamente a exceção reformulada condicionalmente.
    throw $exception;

    // Como alternativa, lide com o erro, ex.: defina uma mensagem flash e
    // redirecione para a versão HTTPS da página solicitada.
}
```

O parâmetro `$type` pode ter os seguintes valores:

- "auth" Indica um erro de validação do formulário ou um erro de incompatibilidade de controlador/ação.
- "secure" Indica uma falha de restrição do método SSL.

Restringir Ações ao SSL

`SecurityComponent::requireSecure`()

Define as ações que requerem uma solicitação protegida por SSL. Leva qualquer número de argumentos. Pode ser chamado sem argumentos para forçar todas as ações a exigir um SSL protegido.

`SecurityComponent::requireAuth`()

Define as ações que requerem um token válido gerado pelo Componente de segurança. Leva qualquer número de argumentos. Pode ser chamado sem argumentos para forçar todas as ações a exigir uma autenticação válida.

Restringindo a Comunicação entre Controladores

allowedControllers
: Uma lista de controladores que podem enviar solicitações para esse controlador. Isso pode ser usado para controlar solicitações entre controladores.

allowedActions
: Uma lista de ações que têm permissão para enviar solicitações para as ações deste controlador. Isso pode ser usado para controlar solicitações entre controladores.

Essas opções de configuração permitem restringir a comunicação entre controladores.

Prevenção de Adulteração de Formulários

Por padrão, o `SecurityComponent` impede que os usuários adulterem formulários de maneiras específicas. O `` SecurityComponent`` impedirá o seguinte:

- Campos desconhecidos não podem ser adicionados ao formulário.
- Os campos não podem ser removidos do formulário.
- Os valores nas entradas ocultas não podem ser modificados.

A prevenção desses tipos de adulteração é realizada trabalhando com o `FormHelper` e rastreando quais campos estão em um formulário. Os valores para campos ocultos também são rastreados. Todos esses dados são combinados e transformados em um hash. Quando um formulário é enviado, o `SecurityComponent` usará os dados do POST para criar a mesma estrutura e comparar o hash.

Nota: O SecurityComponent **não** impede que as opções selecionadas sejam adicionadas/alteradas. Nem impedirá que as opções de rádio sejam adicionadas/alteradas.

---

unlockedFields
: Defina para uma lista de campos de formulário a serem excluídos da validação do POST. Os campos podem ser desbloqueados no Component ou com `FormHelper::unlockField()`. Os campos que foram desbloqueados não precisam fazer parte do POST e os campos desbloqueados ocultos não têm seus valores verificados.

validatePost
: Defina como `false` para ignorar completamente a validação de solicitações POST, essencialmente desativando a validação de formulário.

Uso

Geralmente, o uso do componente de segurança é feito no `beforeFilter()` do controlador. Você especificaria as restrições de segurança que deseja e o Componente de Segurança as aplicará em sua inicialização:

```
namespace App\Controller;

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

class WidgetsController extends AppController
{
    public function initialize(): void
    {
```

(continues on next page)

(continuação da página anterior)

```
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }
}
```

O exemplo acima forçaria todas as ações que tinham roteamento de administrador a exigir solicitações SSL seguras:

```
namespace App\Controller;

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

class WidgetsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Security', ['blackHoleCallback' => 'forceSSL']);
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        if ($this->request->getParam('admin')) {
            $this->Security->requireSecure();
        }
    }

    public function forceSSL($error = '', SecurityException $exception = null)
    {
        if ($exception instanceof SecurityException && $exception->getType() === 'secure
↪') {
            return $this->redirect('https://' . env('SERVER_NAME') . Router::url($this->
↪request->getRequestTarget()));
        }

        throw $exception;
    }
}
```

Este exemplo forçaria todas as ações que tinham roteamento de administrador a exigir solicitações SSL seguras. Quando a solicitação é ocultada em preto, ele chama o retorno de chamada `forceSSL()`, que redirecionará solicitações não seguras para proteger solicitações automaticamente.

Proteção CSRF

CSRF ou falsificação de solicitação entre sites é uma vulnerabilidade comum em aplicativos da web. Ele permite que um invasor capture e reproduza uma solicitação anterior e, às vezes, envie solicitações de dados usando tags ou recursos de imagem em outros domínios. Para habilitar os recursos de proteção CSRF, use *Falsificação de Solicitação entre Sites (CSRF) Middleware*.

Desabilitando o Componente de Segurança para Ações Específicas

Pode haver casos em que você deseja desativar todas as verificações de segurança de uma ação (por exemplo, solicitações AJAX). Você pode «desbloquear» essas ações listando-as em `$this->Security->unlockedActions` em seu `beforeFilter()`. A propriedade `unlockedActions` **não** afeta outros recursos do `SecurityComponent`:

```
namespace App\Controller;

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

class WidgetController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Security');
    }

    public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        $this->Security->setConfig('unlockedActions', ['edit']);
    }
}
```

Este exemplo desabilitaria todas as verificações de segurança da ação de edição.

Pagination

class Cake\Controller\Component**PaginatorComponent**

Um dos principais obstáculos da criação de aplicativos Web flexíveis e fáceis de usar é o design de uma interface de usuário intuitiva. Muitos aplicativos tendem a crescer em tamanho e complexidade rapidamente, e designers e programadores acham que não conseguem lidar com a exibição de centenas ou milhares de registros. A refatoração leva tempo, e o desempenho e a satisfação do usuário podem sofrer.

A exibição de um número razoável de registros por página sempre foi uma parte crítica de todos os aplicativos e usada para causar muitas dores de cabeça aos desenvolvedores. O CakePHP facilita a carga para o desenvolvedor, fornecendo uma maneira rápida e fácil de paginar os dados.

A paginação no CakePHP é oferecida por um componente no controlador, para facilitar a criação de consultas paginadas. A View *PaginatorHelper* é usada para simplificar a geração de links e botões de paginação.

Usando Controller::paginate()

No controlador, começamos definindo as condições de consulta padrão que a paginação usará na variável do controlador `$paginate`. Essas condições servem como base para suas consultas de paginação. Eles são aumentados pelos parâmetros `sort`, `direction`, `limit` e `page` transmitidos a partir da URL. É importante notar que a chave `order` deve ser definida em uma estrutura de matriz como abaixo:

```
class ArticlesController extends AppController
{
    public $paginate = [
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
}
```

Você também pode incluir qualquer uma das opções suportadas por *find()*, como `fields`:

```
class ArticlesController extends AppController
{
    public $paginate = [
        'fields' => ['Articles.id', 'Articles.created'],
        'limit' => 25,
        'order' => [
            'Articles.title' => 'asc'
        ]
    ];

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
}
```

Enquanto você pode passar a maioria das opções de consulta da propriedade paginate, geralmente é mais fácil e simples agrupar suas opções de paginação em *Personalizando Metódos de Consulta*. Você pode definir o uso da paginação do localizador, definindo a opção `finder`:

```
class ArticlesController extends AppController
{
    public $paginate = [
        'finder' => 'published',
    ];
}
```

Como os métodos localizadores personalizados também podem receber opções, é assim que você passa as opções para

um método find personalizado dentro da propriedade paginate:

```
class ArticlesController extends AppController
{
    // encontrar artigos por tag
    public function tags()
    {
        $tags = $this->request->getParam('pass');

        $customFinderOptions = [
            'tags' => $tags
        ];
        // o método find personalizado é chamado findTagged dentro de ArticlesTable.php,
        // e deve ter se parecer com: public function findTagged(Query $query, array
↪$options) {
        // portanto, você usa tags como a chave
        $this->paginate = [
            'finder' => [
                'tagged' => $customFinderOptions
            ]
        ];
        $articles = $this->paginate($this->Articles);
        $this->set(compact('articles', 'tags'));
    }
}
```

Além de definir valores gerais de paginação, você pode definir mais de um conjunto de padrões de paginação no controlador, basta nomear as chaves da matriz após o modelo que deseja configurar:

```
class ArticlesController extends AppController
{
    public $paginate = [
        'Articles' => [],
        'Authors' => [],
    ];
}
```

Os valores das chaves `Articles` e `Authors` podem conter todas as propriedades que um modelo/chave menos a matriz `$paginate`.

Depois de definida a propriedade `$paginate`, podemos usar o método *paginate()* para criar os dados de paginação e adicionar o `PaginatorHelper `` se ainda não foi adicionado. O método paginado do controlador retornará o conjunto de resultados da consulta paginada e definirá os metadados de paginação para a solicitação. Você pode acessar os metadados da paginação em ``$this->request->getParam('paging')`. Um exemplo mais completo do uso de `paginate()` seria:

```
class ArticlesController extends AppController
{
    public function index()
    {
        $this->set('articles', $this->paginate());
    }
}
```

Por padrão, o método `paginate()` usará o modelo padrão para um controlador. Você também pode passar a consulta

resultante de um método find:

```
public function index()
{
   $query = $this->Articles->find('popular')->where(['author_id' => 1]);
   $this->set('articles', $this->paginate($query));
}
```

Se você quiser paginar um modelo diferente, poderá fornecer uma consulta para ele, sendo o próprio objeto de tabela ou seu nome:

```
// Usando a query.
$comments = $this->paginate($commentsTable->find());

// Usando o nome do modelo.
$comments = $this->paginate('Comments');

// Usando um objeto de tabela.
$comments = $this->paginate($commentTable);
```

Usando o Paginator Diretamente

Se você precisar paginar os dados de outro componente, poderá usar o PaginatorComponent diretamente. O PaginatorComponent possui uma API semelhante ao método do controlador:

```
$articles = $this->Paginator->paginate($articleTable->find(), $config);

// Ou
$articles = $this->Paginator->paginate($articleTable, $config);
```

O primeiro parâmetro deve ser o objeto de consulta de um objeto de localização na tabela do qual você deseja paginar os resultados. Opcionalmente, você pode passar o objeto de tabela e permitir que a consulta seja construída para você. O segundo parâmetro deve ser a matriz de configurações a serem usadas para paginação. Essa matriz deve ter a mesma estrutura que a propriedade `$paginate` em um controlador. Ao paginar um objeto `Query`, a opção `finder` será ignorada. Supõe-se que você esteja passando a consulta que deseja paginar.

Paginando Múltiplas Queries

Você pode paginar vários modelos em uma única ação do controlador, usando a opção `scope` na propriedade `$paginate` do controlador e na chamada para o método `paginate()`:

```
// Propriedade Paginar
public $paginate = [
    'Articles' => ['scope' => 'article'],
    'Tags' => ['scope' => 'tag']
];

// Em um método do controlador
$articles = $this->paginate($this->Articles, ['scope' => 'article']);
$tags = $this->paginate($this->Tags, ['scope' => 'tag']);
$this->set(compact('articles', 'tags'));
```

A opção `scope` resultará na `PaginatorComponent` procurando nos parâmetros da string de consulta com escopo definido. Por exemplo, o URL a seguir pode ser usado para paginar tags e artigos ao mesmo tempo:

```
/dashboard?article[page]=1&tag[page]=3
```

Veja a seção *paginator-helper-multiple* para saber como gerar elementos HTML com escopo e URLs para paginação.

Paginando o Mesmo Modelo Várias Vezes

Para paginar o mesmo modelo várias vezes em uma única ação do controlador, é necessário definir um alias para o modelo. Consulte *table-registry-usage* para obter detalhes adicionais sobre como usar o registro da tabela:

```
// Em um método do controlador
$this->paginate = [
    'ArticlesTable' => [
        'scope' => 'published_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
    'UnpublishedArticlesTable' => [
        'scope' => 'unpublished_articles',
        'limit' => 10,
        'order' => [
            'id' => 'desc',
        ],
    ],
];

// Registre um objeto de tabela adicional para permitir a diferenciação no componente de␣
↪paginação
TableRegistry::getTableLocator()->setConfig('UnpublishedArticles', [
    'className' => 'App\Model\Table\ArticlesTable',
    'table' => 'articles',
    'entityClass' => 'App\Model\Entity\Article',
]);

$publishedArticles = $this->paginate(
    $this->Articles->find('all', [
        'scope' => 'published_articles'
    ])->where(['published' => true])
);

$unpublishedArticles = $this->paginate(
    TableRegistry::getTableLocator()->get('UnpublishedArticles')->find('all', [
        'scope' => 'unpublished_articles'
    ])->where(['published' => false])
);
```

Controlar Quais Campos Usados para Ordenamento

Por padrão, a classificação pode ser feita em qualquer coluna não virtual que uma tabela tenha. Às vezes, isso é indesejável, pois permite que os usuários classifiquem em colunas não indexadas que podem ser caras de solicitar. Você pode definir a lista de permissões dos campos que podem ser classificados usando a opção `sortableFields`. Essa opção é necessária quando você deseja classificar os dados associados ou os campos computados que podem fazer parte da sua consulta de paginação:

```
public $paginate = [
    'sortableFields' => [
        'id', 'title', 'Users.username', 'created'
    ]
];
```

Quaisquer solicitações que tentem classificar campos que não estão na lista de permissões serão ignoradas.

Limitar o Número Máximo de Linhas por Página

O número de resultados que são buscados por página é exposto ao usuário como o parâmetro `limit`. Geralmente, é indesejável permitir que os usuários busquem todas as linhas em um conjunto paginado. A opção `maxLimit` afirma que ninguém pode definir esse limite muito alto do lado de fora. Por padrão, o CakePHP limita o número máximo de linhas que podem ser buscadas para 100. Se esse padrão não for apropriado para a sua aplicação, você poderá ajustá-lo como parte das opções de paginação, por exemplo, reduzindo-o para `10`:

```
public $paginate = [
    // Outras chaves aqui.
    'maxLimit' => 10
];
```

Se o parâmetro de limite da solicitação for maior que esse valor, ele será reduzido ao valor `maxLimit`.

Juntando Associações Adicionais

Associações adicionais podem ser carregadas na tabela paginada usando o parâmetro `contains`:

```
public function index()
{
    $this->paginate = [
        'contain' => ['Authors', 'Comments']
    ];

    $this->set('articles', $this->paginate($this->Articles));
}
```

Solicitações de Página Fora do Intervalo

O PaginatorComponent lançará uma `NotFoundException` ao tentar acessar uma página inexistente, ou seja, o número da página solicitada é maior que a contagem total de páginas.

Portanto, você pode permitir que a página de erro normal seja renderizada ou usar um bloco try catch e executar a ação apropriada quando um `NotFoundException` for capturado:

```
use Cake\Http\Exception\NotFoundException;

public function index()
{
    try {
        $this->paginate();
    } catch (NotFoundException $e) {
        // Faça algo aqui como redirecionar para a primeira ou a última página.
        // $this->request->getParam('paging') fornecerá as informações necessárias.
    }
}
```

Paginação na View

Verifique a documentação *PaginatorHelper* para saber como criar links para navegação de paginação.

Request Handler (Tratamento de Requisições)

class RequestHandlerComponent(*ComponentCollection $collection*, *array $config = []*)

O componente Request Handler é usado no CakePHP para obter informações adicionais sobre as solicitações HTTP feitas para sua aplicação. Você pode usá-lo para ver quais tipos de conteúdo os clientes preferem, analisar automaticamente a entrada da solicitação, definir como os tipos de conteúdo são mapeados para exibir classes ou caminhos de modelo.

Por padrão, o RequestHandler detectará automaticamente solicitações AJAX com base no cabeçalho HTTP `X-Requested-With` que muitas bibliotecas JavaScript usam. Quando usado em conjunto com *Cake\Routing\Router::extensions()*, o RequestHandler muda automaticamente os arquivos de layout e modelo para aqueles que correspondem aos tipos de mídia não HTML. Além disso, se existir um auxiliar com o mesmo nome que a extensão solicitada, ele será adicionado à matriz Auxiliar de Controladores. Por fim, se os dados XML/JSON forem enviados para seus controladores, eles serão analisados em uma matriz atribuída a `$this->request->getData()` e, e pode ser acessado como faria com os dados POST padrão. Para fazer uso do RequestHandler, ele deve ser incluído no seu método `initialize()`:

```
class WidgetsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    // Resto do controller
}
```

Obtenção de Informações de Solicitação

O manipulador de solicitações possui vários métodos que fornecem informações sobre o cliente e sua solicitação.

`RequestHandlerComponent::accepts`(*$type = null*)

$type pode ser uma string, uma matriz ou nulo. Se uma string, `accept()` retornará `true` se o cliente aceitar o tipo de conteúdo. Se uma matriz for especificada, `accept()` retorna `true` se qualquer um dos tipos de conteúdo for aceito pelo cliente. Se null, retorna uma matriz dos tipos de conteúdo que o cliente aceita. Por exemplo:

```
class ArticlesController extends AppController
{

    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function beforeFilter(EventInterface $event)
    {
        if ($this->RequestHandler->accepts('html')) {
            // Execute o código apenas se o cliente aceitar uma resposta
            // HTML (text/html).
        } elseif ($this->RequestHandler->accepts('xml')) {
            // Executar código somente XML
        }
        if ($this->RequestHandler->accepts(['xml', 'rss', 'atom'])) {
            // Executa se o cliente aceita qualquer um dos itens acima: XML, RSS
            // ou Atom.
        }
    }
}
```

Outros métodos de detecção de "type" de solicitação incluem:

`RequestHandlerComponent::isXml`()

Retorna `true` se a solicitação atual aceitar XML como resposta.

`RequestHandlerComponent::isRss`()

Retorna `true` se a solicitação atual aceitar RSS como resposta.

`RequestHandlerComponent::isAtom`()

Retorna `true` se a chamada atual aceitar uma resposta Atom, caso contrário, false.

`RequestHandlerComponent::isMobile`()

Retorna `true` se a sequência do agente do usuário corresponder a um navegador da Web móvel ou se o cliente aceitar conteúdo WAP. As sequências suportadas do Mobile User Agent são:

- Android
- AvantGo
- BlackBerry
- DoCoMo
- Fennec

- iPad
- iPhone
- iPod
- J2ME
- MIDP
- NetFront
- Nokia
- Opera Mini
- Opera Mobi
- PalmOS
- PalmSource
- portalmmm
- Plucker
- ReqwirelessWeb
- SonyEricsson
- Symbian
- UP.Browser
- webOS
- Windows CE
- Windows Phone OS
- Xiino

`RequestHandlerComponent::isWap()`

Retorna `true` se o cliente aceitar conteúdo WAP.

Todos os métodos de detecção de solicitação acima podem ser usados de maneira semelhante para filtrar a funcionalidade destinada a tipos de conteúdo específicos. Por exemplo, ao responder a solicitações AJAX, geralmente você deseja desativar o cache do navegador e alterar o nível de depuração. No entanto, você deseja permitir o armazenamento em cache para solicitações não-AJAX. O seguinte exemplo faria isso:

```
if ($this->request->is('ajax')) {
    $this->response->disableCache();
}
// Continua ação do controlador
```

Decodificação Automática de Dados de Solicitação

Adicione um decodificador de dados de solicitação. O manipulador deve conter um retorno de chamada e quaisquer argumentos adicionais para o retorno de chamada. O retorno de chamada deve retornar uma matriz de dados contidos na entrada da solicitação. Por exemplo, adicionar um manipulador de CSV pode parecer:

```
class ArticlesController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $parser = function ($data) {
            $rows = str_getcsv($data, "\n");
            foreach ($rows as &$row) {
                $row = str_getcsv($row, ',');
            }

            return $rows;
        };
        $this->loadComponent('RequestHandler', [
            'inputTypeMap' => [
                'csv' => [$parser]
            ]
        ]);
    }
}
```

Você pode usar qualquer callable[69] para a função de manipulação. Você também pode passar argumentos adicionais para o retorno de chamada, isso é útil para retornos de chamada como `json_decode`:

```
$this->RequestHandler->setConfig('inputTypeMap.json', ['json_decode', true]);
```

O exemplo acima tornará `$this->request->getData()` uma matriz dos dados de entrada JSON, sem o `true` adicional, você obteria um conjunto de objetos `stdClass`.

Alterado na versão 3.6.0: Você deve preferir usar *Body Parser Middleware* em vez de RequestHandlerComponent.

VerificandoPreferências de Tipo de Conteúdo

`RequestHandlerComponent::prefers`(*$type = null*)

Determina quais tipos de conteúdo o cliente prefere. Se nenhum parâmetro for fornecido, o tipo de conteúdo mais provável será retornado. Se $type for uma matriz, o primeiro tipo aceito pelo cliente será retornado. A preferência é determinada principalmente pela extensão do arquivo analisada pelo roteador, se houver, e por uma lista de tipos de conteúdo em `HTTP_ACCEPT`:

```
$this->RequestHandler->prefers('json');
```

[69] https://php.net/callback

Respondendo a Solicitações

`RequestHandlerComponent::renderAs`(*$controller*, *$type*)

Altere o modo de renderização de um controlador para o tipo especificado. Também anexará o auxiliar apropriado à matriz auxiliar do controlador, se disponível e ainda não estiver na matriz:

```
// Force o controlador a renderizar uma resposta xml.
$this->RequestHandler->renderAs($this, 'xml');
```

Este método também tentará adicionar um auxiliar que corresponda ao seu tipo de conteúdo atual. Por exemplo, se você renderizar como `rss`, o `RssHelper` será adicionado.

`RequestHandlerComponent::respondAs`(*$type*, *$options*)

Define o cabeçalho da resposta com base nos nomes do mapa do tipo de conteúdo. Este método permite definir várias propriedades de resposta de uma só vez:

```
$this->RequestHandler->respondAs('xml', [
    // Força o download
    'attachment' => true,
    'charset' => 'UTF-8'
]);
```

`RequestHandlerComponent::responseType`()

Retorna o tipo de resposta atual com o Cabeçalho do tipo de conteúdo ou nulo se ainda não tiver sido definido.

Aproveitando a Validação de Cache HTTP

O modelo de validação de cache HTTP é um dos processos usados para gateways de cache, também conhecidos como proxies reversos, para determinar se eles podem servir uma cópia armazenada de uma resposta ao cliente. Sob esse modelo, você economiza principalmente largura de banda, mas, quando usado corretamente. Também é possível economizar algum processamento da CPU, reduzindo assim os tempos de resposta.

A ativação do RequestHandlerComponent no seu controlador ativa automaticamente uma verificação feita antes de renderizar a exibição. Essa verificação compara o objeto de resposta com a solicitação original para determinar se a resposta não foi modificada desde a última vez que o cliente solicitou.

Se a resposta for avaliada como não modificada, o processo de renderização da visualização será interrompido, economizando tempo de processamento, economizando largura de banda e nenhum conteúdo será retornado ao cliente. O código de status da resposta é então definido como `304 Not Modified`.

Você pode desativar essa verificação automática definindo a configuração `checkHttpCache` como `false`:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'checkHttpCache' => false
    ]);
}
```

Usando ViewClasses Customizadas

Ao usar o JsonView/XmlView, você pode substituir a serialização padrão por uma classe View personalizada ou adicionar classes View para outros tipos.

Você pode mapear tipos novos e existentes para suas classes personalizadas. Você também pode definir isso automaticamente usando a configuração `viewClassMap`:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('RequestHandler', [
        'viewClassMap' => [
            'json' => 'ApiKit.MyJson',
            'xml' => 'ApiKit.MyXml',
            'csv' => 'ApiKit.Csv'
        ]
    ]);
}
```

Configurando Componentes

Muitos dos componentes principais requerem configuração. Alguns exemplos de componentes que requerem configuração são *Security (Segurança)* e *Request Handler (Tratamento de Requisições)*. A configuração desses componentes e dos componentes em geral é geralmente feita via `loadComponent()` no método `initialize()` do seu Controlador ou através do array `$components`:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('RequestHandler', [
            'viewClassMap' => ['json' => 'AppJsonView'],
        ]);
        $this->loadComponent('Security', ['blackholeCallback' => 'blackhole']);
    }

}
```

Você pode configurar componentes em tempo de execução usando o método `setConfig()`. Muitas vezes, isso é feito no método `beforeFilter()` do seu controlador. O exemplo acima também pode ser expresso como:

```
public function beforeFilter(EventInterface $event)
{
    $this->RequestHandler->setConfig('viewClassMap', ['rss' => 'MyRssView']);
}
```

Como os auxiliares, os componentes implementam os métodos `getConfig()` e `setConfig()` para ler e gravar dados de configuração:

```
// Leia os dados de configuração.
$this->RequestHandler->getConfig('viewClassMap');
```

(continues on next page)

(continuação da página anterior)

```
// Definir configuração
$this->Csrf->setConfig('cookieName', 'token');
```

Assim como os auxiliares, os componentes mesclam automaticamente sua propriedade `$ _defaultConfig` com a configuração do construtor para criar a propriedade `$_config` que pode ser acessada com `getConfig()` e `setConfig()`.

Alias em Componentes

Uma configuração comum a ser usada é a opção `className`, que permite o alias de componentes. Esse recurso é útil quando você deseja substituir `$this->Auth` ou outra referência de componente comum por uma implementação personalizada:

```
// src/Controller/PostsController.php
class PostsController extends AppController
{
    public function initialize(): void
    {
        $this->loadComponent('Auth', [
            'className' => 'MyAuth'
        ]);
    }
}

// src/Controller/Component/MyAuthComponent.php
use Cake\Controller\Component\AuthComponent;

class MyAuthComponent extends AuthComponent
{
    // Adicione seu código para substituir o principal AuthComponent
}
```

O exemplo acima seria *alias* `MyAuthComponent` para `$this->Auth` em seus controladores.

Nota: O alias de um componente substitui essa instância em qualquer lugar em que esse componente seja usado, inclusive dentro de outros componentes.

Carregando Componentes em Tempo Real

Você pode não precisar de todos os seus componentes disponíveis em todas as ações do controlador. Em situações como essa, você pode carregar um componente em tempo de execução usando o método `loadComponent()` no seu controlador:

```
// Em um método do controlador
$this->loadComponent('OneTimer');
$time = $this->OneTimer->getTime();
```

Nota: Lembre-se de que os componentes carregados em tempo real não terão retornos de chamada perdidos. Se você confiar nos retornos de chamada `beforeFilter` ou `startup` que estão sendo chamados, pode ser necessário chamá-los manualmente, dependendo de quando você carregar o componente.

Usando Componentes

Depois de incluir alguns componentes no seu controlador, usá-los é bastante simples. Cada componente usado é exposto como uma propriedade no seu controlador. Se você carregou a classe *Cake\Controller\Component\FlashComponent* no seu controlador, é possível acessá-lo da seguinte maneira:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Flash');
    }

    public function delete()
    {
        if ($this->Post->delete($this->request->getData('Post.id')) {
            $this->Flash->success('Post deleted.');

            return $this->redirect(['action' => 'index']);
        }
    }
```

Nota: Como os Modelos e os Componentes são adicionados aos Controladores como propriedades, eles compartilham o mesmo "namespace". Certifique-se de não dar o mesmo nome a um componente de um modelo.

Criando um Componente

Suponha que nosso aplicativo precise executar uma operação matemática complexa em muitas partes diferentes do aplicativo. Poderíamos criar um componente para hospedar essa lógica compartilhada para uso em muitos controladores diferentes.

O primeiro passo é criar um novo arquivo e classe de componente. Crie o arquivo em **src/Controller/Component/MathComponent.php**. A estrutura básica do componente será semelhante a isso:

```
namespace App\Controller\Component;

use Cake\Controller\Component;

class MathComponent extends Component
{
    public function doComplexOperation($amount1, $amount2)
    {
        return $amount1 + $amount2;
```

(continues on next page)

(continuação da página anterior)

```
    }
}
```

Nota: Todos os componentes devem estender `Cake\Controller\Component`. Não fazer isso acionará uma exceção.

Incluindo seu Componente em seus Controladores

Depois que nosso componente é concluído, podemos usá-lo nos controladores do aplicativo carregando-o durante o método `initialize()` do controlador. Uma vez carregado, o controlador receberá um novo atributo com o nome do componente, através do qual podemos acessar uma instância dele:

```
// Em um controlador
// Disponibilize o novo componente em $this->Math,
// bem como o padrão $this->Csrf
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math');
    $this->loadComponent('Csrf');
}
```

Ao incluir componentes em um controlador, você também pode declarar um conjunto de parâmetros que serão passados para o construtor do componente. Esses parâmetros podem ser manipulados pelo componente:

```
// Em seu controlador
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math', [
        'precision' => 2,
        'randomGenerator' => 'srand'
    ]);
    $this->loadComponent('Csrf');
}
```

O exemplo acima passaria um array contendo precision e randomGenerator para `MathComponent::initialize()` no parâmetro `$config`.

Usando Outros Componentes em seu Componente

Às vezes, um de seus componentes pode precisar usar outro componente. Nesse caso, você pode incluir outros componentes no seu componente exatamente da mesma maneira que os inclui nos controladores - usando o atributo `$components`:

```
// src/Controller/Component/CustomComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;
```

(continues on next page)

(continuação da página anterior)

```
class CustomComponent extends Component
{
    // O outro componente que seu componente usa
    public $components = ['Existing'];

    // Execute qualquer outra configuração adicional para o seu componente.
    public function initialize(array $config): void
    {
        $this->Existing->foo();
    }

    public function bar()
    {
        // ...
    }
}

// src/Controller/Component/ExistingComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class ExistingComponent extends Component
{

    public function foo()
    {
        // ...
    }
}
```

Nota: Ao contrário de um componente incluído em um controlador, nenhum retorno de chamada será acionado no componente de um componente.

Acessando o Controlador de um Componente

De dentro de um componente, você pode acessar o controlador atual através do registro:

```
$controller = $this->_registry->getController();
```

Você pode acessar o controlador em qualquer método de retorno de chamada do objeto de evento:

```
$controller = $event->getSubject();
```

Callback de Componentes

Os componentes também oferecem alguns retornos de chamada do ciclo de vida da solicitação que permitem aumentar o ciclo da solicitação.

beforeFilter(*EventInterface $event*)

É chamado antes do método beforeFilter do controlador, mas *após* o método initialize() do controlador.

startup(*EventInterface $event*)

É chamado após o método beforeFilter do controlador, mas antes que o controlador execute o manipulador de ações atual.

beforeRender(*EventInterface $event*)

É chamado após o controlador executar a lógica da ação solicitada, mas antes de o controlador renderizar visualizações e layout.

shutdown(*EventInterface $event*)

É chamado antes que a saída seja enviada ao navegador.

beforeRedirect(*EventInterface $event*, *$url*, *Response $response*)

É chamado quando o método de redirecionamento do controlador é chamado, mas antes de qualquer ação adicional. Se esse método retornar `false`, o controlador não continuará redirecionando a solicitação. Os parâmetros $url e $response permitem inspecionar e modificar o local ou qualquer outro cabeçalho na resposta.

CAPÍTULO 11

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 camanda View.

O CakePHP vem com a 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` extende 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
<?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 **.php** (CakePHP Template) e utiliza a Sintaxe PHP alternativa[70] 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
<?php echo $variable; ?>
```

Usando o suporte a Tag curta:

```php
<?= $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 substituida 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 virgula` depois da estrutura do foreach (Exceto o último), existem `dois pontos`.

O bloco a seguir é um exempo do uso de `foreach`:

```
<ul>
<?php foreach ($todo as $item): ?>
```

(continues on next page)

[70] https://php.net/manual/en/control-structures.alternative-syntax.php

(continuação da página anterior)

```
  <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>
<?php else: ?>
   <h3>Olá usuário desconhecido</h3>
<?php endif; ?>
```

Se você preferir usar uma linguagem de template como Twig[71], uma subclasse da *View* irá ligar sua linguagem de template e o CakePHP.

Arquivos de template são armazenados em **templates/**, 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 **templates/Products/view.php**.

A camada *view* do CakePHP pode ser constituida por um número diferente de partes. Cada parte tem diferentes usos, e serão abordadas nesse capítulo:

- **templates**: 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 reúsaveis do código da *view*. *Elements* são usualmente renderizados dentro das *views*.
- **layouts**: Arquivos de modelo que contem 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 conponentes 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); ?>
```

[71] https://twig.sensiolabs.org

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".

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* especifica 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:

```
<!-- templates/Common/view.php -->
<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 extendida. Assumindo que nosso arquivo da *view* tem a variável `$post` com os dados sobre nosso *post*, a view poderia se parecer com isso:

```
<!-- templates/Posts/view.php -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post->title);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', [
    'action' => 'edit',
    $post->id
]); ?>
</li>
<?php $this->end(); ?>
```

(continues on next page)

(continuação da página anterior)

```
// 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.php** sendo renderizada como a *view* para a view atual.

Você pode aninhar as view estendidas quantas vezes achar necessário. Cada *view* pode extender 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 usand 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/inicio 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();
```

(continues on next page)

(continuação da página anterior)

```
// 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* expecifico.:

```
// Limpa o conteúdo anterior do *block* sidebar.
$this->reset('sidebar');

// Atribuir uma string vazia também limpará o bloco *sidebar*.
$this->assign('sidebar', '');
```

Atribuir um conteúdo de um *block* muitas vezes é usado quando você quer converter uma varável 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 titulos ou outras marcações:

```
// In templates/layout/default.php
<?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 **templates/layout/default.php**. 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 **templates/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>
```

(continues on next page)

(continuação da página anterior)

```
<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 **templates/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
```

(continues on next page)

(continuação da página anterior)

```
    $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');
```

(continues on next page)

(continuação da página anterior)

```
        // 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 **templates/element/**, e tem a extensão .php. 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 **templates/element/helpbox.php** pode usar a variável `$helptext`:

```
// Dentro do arquivo templates/element/helpbox.php
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
```

(continues on next page)

(continuação da página anterior)

```
    ]
);
```

Cache de Elementos é facilidatado 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 dinamicos 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.php**), use o seguinte:

```
echo $this->element('Contacts.sidebar/helpbox');
```

Routing prefix e *Elements*

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 vocẽ chama:

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

O primeiro *element* procurado será em **templates/Admin/Element/**. Se o arquivo não existir, será procurado na localizaço 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[72], 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 **templates/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*.
- 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 **templates/cell/Inbox/display.php**. 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.

[72] https://github.com/trailblazer/cells

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:

```
<!-- templates/cell/Inbox/display.php -->
<div class="notification-icon">
    Você tem <?= $unread_count ?> mensagen 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 métodos 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 parametros 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 minuscula e separada com underscore do nome da nossa action, e.g. **display.php**.

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 remomendado 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 especifica 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 variaveis 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 dua página rapidamente. Além de arquivos de templates, eles também podem provers *helpers* e "cells" se o seu tema assim requerer. Quando usado `cells` e `helpes` 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';
```

(continues on next page)

(continuação da página anterior)

```
    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 tempalte 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 **templates/**.

Por exemplo, o arquivo de exibição para uma *action* de edição de um `controller` de posts residiria em **plugins/Modern/templates/Posts/edit.php**. Os arquivos de layout residiriam em **plugins/Modern/templates/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/templates/plugin/Cms/Tags/edit.php** 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 **templates/**. 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[73] 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[74] 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 = []*)

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[75] 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[76] 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.

[73] https://github.com/cakephp/docs
[74] https://github.com/cakephp/docs
[75] https://github.com/cakephp/docs
[76] 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, sinta-se a vontade para nos enviar um *pull request* para o Github[77] 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, sinta-se a vontade para nos enviar um *pull request* para o Github[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.

Number

class Cake\View\Helper**NumberHelper**(*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[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.

---

[77] https://github.com/cakephp/docs
[78] https://github.com/cakephp/docs
[79] https://github.com/cakephp/docs

Paginator

`class` Cake\View\Helper**PaginatorHelper**(*View $view*, *array $config = []*)

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[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.

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[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.

Time

`class` Cake\View\Helper**TimeHelper**(*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[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.

---

[80] https://github.com/cakephp/docs
[81] https://github.com/cakephp/docs
[82] https://github.com/cakephp/docs

Url

class Cake\View\Helper**UrlHelper**(*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[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.

Classe *Helper*

[83] https://github.com/cakephp/docs

CAPÍTULO 12

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 extendido 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*.

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;

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();
foreach ($query as $row) {
```

(continues on next page)

(continuação da página anterior)

```
    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 insera classes básicas enquanto sua aplicação não criou uma classe concreta. Se quiséssemos customizar nossa classe ArticlesTable adicionando algumas associações ou definir alguns métodos adicionais, deveriamos 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 `TableRegistry` como antes:

```
use Cake\ORM\TableRegistry;

// Agora $articles é uma instância de nossa classe ArticlesTable.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();

foreach ($query as $row) {
```

(continues on next page)

(continuação da página anterior)

```
    // 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 auxilio 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 atravéz do plugin da comunidade Driver para Banco de Dados Oracle[84].

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');
```

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']);
```

[84] https://github.com/CakeDC/cakephp-oracle-driver

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 excluír 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' => 'secret',
        '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',
```

(continues on next page)

(continuação da página anterior)

```
    'password' => 'secret',
    '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:secret@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;

$connection = 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 definadas em seus arquivos de configuração em tempo de execução:

```
ConnectionManager::config('my_connection', $config);
$connection = 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**TypeFactory**

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 horario 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 horario 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[85].

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.

Tipo DateTime

`class` Cake\Database**DateTimeType**

Mapas para um tipo de coluna nativa `DATETIME`. No PostgreSQL e SQL Server isto se transforma em um tipo `TIMESTAMP`. O valor de retorno padrão deste tipo de coluna é `Cake\I18n\FrozenTime` que estende a classe `DateTimeImmutable` embutida e Chronos[86].

Cake\Database\DateTimeType::**setTimezone**(*string|\DateTimeZone|null $timezone*)

Se o fuso horário de seu servidor de banco de dados não corresponder ao fuso horário PHP de sua aplicação então você pode usar este método para especificar o fuso horário de seu banco de dados. Este fuso horário será então utilizado na conversão de objetos PHP para a cadeia de data/hora do banco de dados e vice versa.

[85] https://github.com/cakephp/chronos
[86] https://github.com/cakephp/chronos

class Cake\Database**DateTimeFractionalType**

Pode ser utilizado para mapear colunas de data/hora que contenham microssegundos como `DATETIME(6)` no MySQL. Para utilizar este tipo você precisa adicioná-lo como um tipo mapeado:

```
// em config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeFractionalType;

// Sobregravar o tipo de data/hora padrão com uma mais precisa.
TypeFactory::map('datetime', DateTimeFractionalType::class);
```

class Cake\Database**DateTimeTimezoneType**

Pode ser usado para mapear colunas de data/hora que contenham fusos horários, tais como `TIMESTAMPTZ` no PostgreSQL. Para utilizar este tipo você precisa adicioná-lo como um tipo mapeado:

```
// em config/bootstrap.php
use Cake\Database\TypeFactory;
use Cake\Database\Type\DateTimeTimezoneType;

// Sobregravar o tipo de data/hora padrão com uma mais precisa.
TypeFactory::map('datetime', DateTimeTimezoneType::class);
```

Adicionando Tipos Personalizados

class Cake\Database**TypeFactory**

static Cake\Database\TypeFactory::**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\TypeFactory;
use PDO;

class JsonType extends Type
{

    public function toPHP($value, Driver $driver)
```

(continues on next page)

(continuação da página anterior)

```
    {
        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\TypeFactory;

TypeFactory::map('json', 'App\Database\Type\JsonType');
```

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 *getSchema()* da Tabela:

```
use Cake\Database\Schema\TableSchemaInterface;

class WidgetsTable extends Table
{

    public function getSchema(): TableSchemaInterface
    {
```

(continues on next page)

(continuação da página anterior)

```
        $schema = parent::getSchema();
        $schema->columnType('widget_prefs', 'json');

        return $schema;
    }

}
```

Mapeando Tipos de Dados Personalizados para Expressões SQL

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:

```
// no src/Database/Point.php
namespace App\Database;

// Nosso objeto de valor é imutável.
class Point
{
    protected $_lat;
    protected $_long;

    // Método de fábrica.
    public static function parse($value)
    {
        // Analise os dados WKB do MySQL.
        $unpacked = unpack('x4/corder/Ltype/dlat/dlong', $value);

        return new static($unpacked['lat'], $unpacked['long']);
    }

    public function __construct($lat, $long)
    {
        $this->_lat = $lat;
        $this->_long = $long;
    }

    public function lat()
    {
        return $this->_lat;
    }

    public function long()
    {
        return $this->_long;
    }
```

(continues on next page)

(continuação da página anterior)

```
}
```

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\DriverInterface;
use Cake\Database\Expression\FunctionExpression;
use Cake\Database\ExpressionInterface;
use Cake\Database\Type\BaseType;
use Cake\Database\Type\ExpressionTypeInterface;

class PointType extends BaseType implements ExpressionTypeInterface
{
    public function toPHP($value, DriverInterface $d)
    {
        return Point::parse($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($value): ExpressionInterface
    {
        if ($value instanceof Point) {
            return new FunctionExpression(
                'POINT',
                [
                    $value->lat(),
                    $value->long()
                ]
            );
        }
        if (is_array($value)) {
            return new FunctionExpression('POINT', [$value[0], $value[1]]);
        }
        // Lidar com outros casos.
    }

    public function toDatabase($value, DriverInterface $driver)
    {
```

(continues on next page)

(continuação da página anterior)

```
        return $value;
    }
}
```

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 *connectar nosso tipo personalizado à nossa classe table*.

Habilitando Objetos DateTime Imutáveis

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:

```
TypeFactory::build('datetime')->useImmutable();
TypeFactory::build('date')->useImmutable();
TypeFactory::build('time')->useImmutable();
TypeFactory::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 quererá 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:

```
$statement = $connection->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:

```
$statement = $connection->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:

```
$statement = $connection->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 = $connection->newQuery();
$query->update('articles')
    ->set(['published' => true])
    ->where(['id' => 2]);
$statement = $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 = $connection->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:

```
$connection->begin();
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
$connection->execute('UPDATE articles SET published = ? WHERE id = ?', [false, 4]);
$connection->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:

```
$connection->transactional(function ($connection) {
    $connection->execute('UPDATE articles SET published = ? WHERE id = ?', [true, 2]);
    $connection->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.
$statement = $connection->execute(
    'SELECT * FROM articles WHERE published = ?',
    [true]
);

// Instruções do ``prepare``serão parâmetros para placeholders.
// Você precisa vincular os parâmetros antes de executar.
$statement = $connection->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`:

```
$statement = $connection->prepare(
    'SELECT * FROM articles WHERE published = ? AND created > ?'
);
// Vincular vários valores
$statement->bind(
    [true, new DateTime('2013-01-01')],
    ['boolean', 'date']
);

// Vincular único valor
$statement->bindValue(1, true, 'boolean');
$statement->bindValue(2, new DateTime('2013-01-01'), 'date');
```

Ao criar instruções, você também pode usar chave de array nomeadas em vez de posicionais:

```
$statement = $connection->prepare(
    'SELECT * FROM articles WHERE published = :published AND created > :created'
);

// Vincular vários valores
$statement->bind(
    ['published' => true, 'created' => new DateTime('2013-01-01')],
    ['published' => 'boolean', 'created' => 'date']
);

// Vincular um valor único
$statement->bindValue('published', true, 'boolean');
$statement->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:

```
$statement->execute();

// Lê uma linha.
$row = $statement->fetch('assoc');

// Lê todas as linhas.
```

(continues on next page)

(continuação da página anterior)

```
$rows = $statement->fetchAll('assoc');

// Lê linhas através de iteração.
foreach ($statement 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:

```
$rowCount = count($statement);
$rowCount = $statement->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 = $statement->errorCode();
$info = $statement->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.
$connection->logQueries(true);

// Desabilita o log de consultas.
$connection->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',
```

(continues on next page)

(continuação da página anterior)

```
    '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:

```
$connection->getDriver()->enableAutoQuoting();
```

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',
```

(continues on next page)

(continuação da página anterior)

```
    ]
],
```

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

class Cake\ORM**Query**

O construtor de consultas do ORM fornece uma interface fluente e simples de usar para criar e executar consultas. Ao compor consultas, você pode criar consultas avançadas usando uniões e subconsultas com facilidade.

Debaixo do capô, o construtor de consultas usa instruções preparadas para DOP que protegem contra ataques de injeção de SQL.

O Objeto Query

A maneira mais fácil de criar um objeto `Consulta` é usar `find()` de um objeto `Table`. Este método retornará uma consulta incompleta pronta para ser modificada. Também é possível usar o objeto de conexão de uma tabela para acessar o construtor Query de nível inferior que não inclui recursos ORM, se necessário. Consulte a seção *Executando Consultas* para obter mais informações:

```
use Cake\ORM\TableRegistry;

// Anterior a 3.6.0
$articles = TableRegistry::get('Articles');

$articles = TableRegistry::getTableLocator()->get('Articles');

// Inicie uma nova consulta.
$query = $articles->find();
```

Quando dentro de um controlador, você pode usar a variável de tabela automática criada usando o sistema de convenções:

```
// Dentro de ArticlesController.php

$query = $this->Articles->find();
```

Selecionando Linhas de uma Tabela

```
use Cake\ORM\TableRegistry;

// Anterior a 3.6.0
$query = TableRegistry::get('Articles')->find();

$query = TableRegistry::getTableLocator()->get('Articles')->find();

foreach ($query as $article) {
    debug($article->title);
}
```

Para os exemplos restantes, suponha que `$articles` seja um *Table*. Quando dentro de controladores, você pode usar `$this->Articles` em vez de `$articles`.

Quase todos os métodos em um objeto `Query` retornam a mesma consulta, isso significa que os objetos `Query` são preguiçosos e não serão executados a menos que você solicite:

```
$query->where(['id' => 1]); // Retornar o mesmo objeto de consulta
$query->order(['title' => 'DESC']); // Ainda o mesmo objeto, nenhum SQL executado
```

É claro que você pode encadear os métodos que você chama nos objetos de consulta:

```
$query = $articles
    ->find()
    ->select(['id', 'name'])
    ->where(['id !=' => 1])
    ->order(['created' => 'DESC']);
```

(continues on next page)

(continuação da página anterior)

```
foreach ($query as $article) {
    debug($article->created);
}
```

Se você tentar chamar `debug()` em um objeto Query, verá seu estado interno e o SQL que será executado no banco de dados:

```
debug($articles->find()->where(['id' => 1]));

// Saídas
// ...
// 'sql' => 'SELECT * FROM articles where id = ?'
// ...
```

Você pode executar uma consulta diretamente sem precisar usar `foreach` nela. A maneira mais fácil é chamar os métodos `all()` ou `toList()`:

```
$resultsIteratorObject = $articles
    ->find()
    ->where(['id >' => 1])
    ->all();

foreach ($resultsIteratorObject as $article) {
    debug($article->id);
}

$resultsArray = $articles
    ->find()
    ->where(['id >' => 1])
    ->all()
    ->toList();

foreach ($resultsArray as $article) {
    debug($article->id);
}

debug($resultsArray[0]->title);
```

No exemplo acima, `$resultsIteratorObject` será uma instância de `Cake\ORM\ResultSet`, um objeto no qual você pode iterar e aplicar vários métodos de extração e deslocamento.

Freqüentemente, não há necessidade de chamar `all()`, você pode simplesmente iterar o objeto Query para obter seus resultados. Objetos de consulta também podem ser usados diretamente como objeto de resultado; tentar iterar a consulta, chamando `toList()` ou alguns dos métodos herdados de *Collection*, resultará na execução da consulta e nos resultados retornados a você.

Selecionando uma Única Linha de uma Tabela

Você pode usar o método `first()` para obter o primeiro resultado na consulta:

```
$article = $articles
    ->find()
    ->where(['id' => 1])
    ->first();

debug($article->title);
```

Obtendo uma Lista de Valores de uma Coluna

```
// Use o método extract() da biblioteca de coleções
// Isso executa a consulta também
$allTitles = $articles->find()->extract('title');

foreach ($allTitles as $title) {
    echo $title;
}
```

Você também pode obter uma lista de valores-chave de um resultado da consulta:

```
$list = $articles->find('list');

foreach ($list as $id => $title) {
    echo "$id : $title"
}
```

Para obter mais informações sobre como personalizar os campos usados para preencher a lista, consulte seção *Encontrando Chaves/Pares de Valores*.

As Consultas são Objetos de Coleção

Depois de se familiarizar com os métodos do objeto Query, é altamente recomendável que você visite a seção *Coleção* para melhorar suas habilidades em percorrer os dados com eficiência. Em resumo, é importante lembrar que qualquer coisa que você possa chamar em um objeto Collection, você também pode fazer em um objeto Query:

```
// Use o método combine() da biblioteca de coleções,
// isto é equivalente a find('list')
$keyValueList = $articles->find()->combine('id', 'title');

// Um exemplo avançado
$results = $articles->find()
    ->where(['id >' => 1])
    ->order(['title' => 'DESC'])
    ->map(function ($row) { // map() é um método de coleção, ele executa a consulta
        $row->trimmedTitle = trim($row->title);

        return $row;
    })
```

(continues on next page)

(continuação da página anterior)

```
    ->combine('id', 'trimmedTitle') // combine() é outro método de coleção
    ->toArray(); // Também um método dA biblioteca de coleções

foreach ($results as $id => $trimmedTitle) {
    echo "$id : $trimmedTitle";
}
```

As Consultas são Avaliadas Preguiçosamente

Objetos de consulta são avaliados preguiçosamente. Isso significa que uma consulta não é executada até que ocorra uma das seguintes coisas:

- A consulta é iterada com `foreach()`.
- O método `execute()` da consulta é chamado. Isso retornará o objeto subjacente de instrução e deve ser usado com consultas de inserção/atualização/exclusão.
- O método `first()` da consulta é chamado. Isso retornará o primeiro resultado no conjunto construído por `SELECT` (ele adiciona `LIMIT 1` à consulta).
- O método `all()` da consulta é chamado. Isso retornará o conjunto de resultados e só pode ser usado com instruções `SELECT`.
- O método `toList()` ou `toArray()` da consulta é chamado.

Até que uma dessas condições seja atendida, a consulta pode ser modificada sem que SQL adicional seja enviado ao banco de dados. Isso também significa que, se uma consulta não tiver sido realizada, nenhum SQL é enviado ao banco de dados. Uma vez executada, modificar e reavaliar uma consulta resultará na execução de SQL adicional.

Se você quiser dar uma olhada no que o SQL CakePHP está gerando, você pode ativar o banco de dados *query logging*.

Selecionando Dados

O CakePHP simplifica a construção de consultas `SELECT`. Para limitar os campos buscados, você pode usar o método `select()`:

```
$query = $articles->find();
$query->select(['id', 'title', 'body']);
foreach ($query as $row) {
    debug($row->title);
}
```

Você pode definir aliases para campos fornecendo campos como uma matriz associativa:

```
// Resultados do SELECT id AS pk, title AS aliased_title, body ...
$query = $articles->find();
$query->select(['pk' => 'id', 'aliased_title' => 'title', 'body']);
```

Para selecionar campos distintos, você pode usar o método `distinct()`:

```
// Resultados em SELECT DISTINCT country FROM ...
$query = $articles->find();
$query->select(['country'])
    ->distinct(['country']);
```

Para definir algumas condições básicas, você pode usar o método `where()`:

```
// As condições são combinadas com AND
$query = $articles->find();
$query->where(['title' => 'First Post', 'published' => true]);

// Você pode chamar where() várias vezes
$query = $articles->find();
$query->where(['title' => 'First Post'])
    ->where(['published' => true]);
```

Você também pode passar uma função anônima para o método `where()`. A função anônima transmitida receberá uma instância de `\Cake\Database\Expression\QueryExpression` como seu primeiro argumento e `\Cake\ORM\Query` como seu segundo argumento:

```
$query = $articles->find();
$query->where(function (QueryExpression $exp, Query $q) {
    return $exp->eq('published', true);
});
```

Veja a seção *Condições Avançadas* para descobrir como construir condições mais complexas com `WHERE`. Para aplicar ordenamentos, você pode usar o método `order`:

```
$query = $articles->find()
    ->order(['title' => 'ASC', 'id' => 'ASC']);
```

Ao chamar `order()` várias vezes em uma consulta, várias cláusulas serão anexadas. No entanto, ao usar finders, às vezes você pode sobrescrever o `ORDER BY`. Defina o segundo parâmetro de `order()` (assim como `orderAsc()` ou `orderDesc()`) como `Query::OVERWRITE` ou como `true`:

```
$query = $articles->find()
    ->order(['title' => 'ASC']);
// Posteriormente, substitua a cláusula ORDER BY em vez de anexá-la.
$query = $articles->find()
    ->order(['created' => 'DESC'], Query::OVERWRITE);
```

Além de `order`, os métodos `orderAsc` e `` orderDesc`` podem ser usados quando você precisa organizar expressões complexas:

```
$query = $articles->find();
$concat = $query->func()->concat([
    'title' => 'identifier',
    'synopsis' => 'identifier'
]);
$query->orderAsc($concat);
```

Para limitar o número de linhas ou definir o deslocamento da linha, você pode usar os métodos `limit()` e `page()`:

```
// Busca linhas de 50 para 100
$query = $articles->find()
    ->limit(50)
    ->page(2);
```

Como você pode ver nos exemplos acima, todos os métodos que modificam a consulta fornecem uma interface fluente, permitindo que você crie uma consulta por meio de chamadas de método em cadeia.

Selecionando Campos Específicos

Por padrão, uma consulta seleciona todos os campos de uma tabela, a exceção é quando você chama a função `select()` e passa determinados campos:

```
// Selecione apenas ID e título da tabela de artigos
$articles->find()->select(['id', 'title']);
```

Se você ainda deseja selecionar todos os campos de uma tabela depois de chamar `select($fields)`, pode passar a instância da tabela para `select()` para esse propósito:

```
// Seleciona todos os campos da tabela de artigos,
// incluindo um campo slug calculado.
$query = $articlesTable->find();
$query
    ->select(['slug' => $query->func()->concat(['title' => 'identifier', '-', 'id' =>
↪'identifier'])])
    ->select($articlesTable); // Select all fields from articles
```

Se você desejar selecionar todos os campos, exceto alguns, em uma tabela, pode usar `selectAllExcept()`:

```
$query = $articlesTable->find();

// Obtenha todos os campos, exceto o campo publicado.
$query->selectAllExcept($articlesTable, ['published']);
```

Você também pode passar um objeto `Association` ao trabalhar com associações embutidas.

Usando Funções SQL

O ORM do CakePHP oferece abstração para algumas funções SQL comumente usadas. O uso da abstração permite que o ORM selecione a implementação específica da plataforma da função desejada. Por exemplo, `concat` é implementado de maneira diferente no MySQL, PostgreSQL e SQL Server. O uso da abstração permite que seu código seja portátil:

```
// Resultados em SELECT COUNT(*) count FROM...
$query = $articles->find();
$query->select(['count' => $query->func()->count('*')]);
```

Várias funções comumente usadas podem ser criadas com o método `func()`:

`rand()`
: Gere um valor aleatório entre 0 e 1 via SQL.

`sum()`
: Calcular uma soma. *Assume que argumentos são valores literais.*

`avg()`
: Calcule uma média. *Assume que argumentos são valores literais.*

`min()`
: Calcule o mínimo de uma coluna. *Assume que argumentos são valores literais.*

`max()`
: Calcule o máximo de uma coluna. *Assume que argumentos são valores literais.*

`count()`
: Calcule a contagem. *Assume que argumentos são valores literais.*

concat()
Concatene dois valores juntos. *Assume que os argumentos são parâmetros vinculados.*

coalesce()
Agrupar valores. *Assume que os argumentos são parâmetros vinculados.*

dateDiff()
Obtenha a diferença entre duas datas/horas. *Assume que os argumentos são parâmetros vinculados.*

now()
O padrão é retornar data e hora, mas aceita "time" ou "date" para retornar apenas esses valores.

extract()
Retorna a parte da data especificada da expressão SQL.

dateAdd()
Adicione a unidade de tempo à expressão de data.

dayOfWeek()
Retorna uma FunctionExpression representando uma chamada para a função SQL WEEKDAY.

Argumentos de Função

Funções SQL chamadas através de `func()` podem aceitar identificadores SQL, valores literais, parâmetros vinculados ou outras instâncias `ExpressionInterface` como argumentos:

```
$query = $articles->find()->innerJoinWith('Categories');
$concat = $query->func()->concat([
    'Articles.title' => 'identifier',
    ' - CAT: ',
    'Categories.name' => 'identifier',
    ' - Age: ',
    $query->func()->dateDiff(
        'NOW()' => 'literal',
        'Articles.created' => 'identifier'
    )
]);
$query->select(['link_title' => $concat]);
```

Os argumentos `literal` e `identifier` permitem que você faça referência a outras colunas e literais SQL enquanto `identifier` será adequadamente citado se a citação automática estiver ativada. Se não marcado como literal ou identificador, os argumentos serão parâmetros vinculados, permitindo que você passe com segurança os dados do usuário para a função.

O exemplo acima gera algo parecido com isto no MYSQL.

```
SELECT CONCAT(
    Articles.title,
    :c0,
    Categories.name,
    :c1,
    (DATEDIFF(NOW(), Articles.created))
) FROM articles;
```

O argumento `:c0` terá o texto `' - CAT:'` quando a consulta for executada. A expressão `dateDiff` foi traduzida para o SQL apropriado.

Funções Customizadas

Se `func()` ainda não envolver a função SQL que você precisa, você poderá chamá-la diretamente através de `func()` e ainda assim passar com segurança argumentos e dados do usuário, conforme descrito. Certifique-se de passar o tipo de argumento apropriado para funções personalizadas ou elas serão tratadas como parâmetros associados:

```
$query = $articles->find();
$year = $query->func()->year([
    'created' => 'identifier'
]);
$time = $query->func()->date_format([
    'created' => 'identifier',
    "'%H:%i'" => 'literal'
]);
$query->select([
    'yearCreated' => $year,
    'timeCreated' => $time
]);
```

Essa função personalizada geraria algo parecido com isto no MYSQL:

```
SELECT YEAR(created) as yearCreated,
       DATE_FORMAT(created, '%H:%i') as timeCreated
FROM articles;
```

Nota: Use `func()` para passar dados não confiáveis do usuário para qualquer função SQL.

Agregadores - Group e Having

Ao usar funções agregadas como `count` e `sum`, você pode usar as cláusulas `group by` e `having`:

```
$query = $articles->find();
$query->select([
    'count' => $query->func()->count('view_count'),
    'published_date' => 'DATE(created)'
])
->group('published_date')
->having(['count >' => 3]);
```

Declarações de Caso

O ORM também oferece a expressão SQL `case`. A expressão `case` permite implementar a lógica `if... then... else` dentro do seu SQL. Isso pode ser útil para gerar relatórios sobre dados nos quais você precisa somar ou contar condicionalmente ou onde precisa de dados específicos com base em uma condição.

Se desejassemos saber quantos artigos publicados estão em nosso banco de dados, poderíamos usar o seguinte SQL:

```
SELECT
COUNT(CASE WHEN published = 'Y' THEN 1 END) AS number_published,
```

(continues on next page)

(continuação da página anterior)

```
COUNT(CASE WHEN published = 'N' THEN 1 END) AS number_unpublished
FROM articles
```

Para fazer isso com o construtor de consultas, usaríamos o seguinte código:

```
$query = $articles->find();
$publishedCase = $query->newExpr()
    ->addCase(
        $query->newExpr()->add(['published' => 'Y']),
        1,
        'integer'
    );
$unpublishedCase = $query->newExpr()
    ->addCase(
        $query->newExpr()->add(['published' => 'N']),
        1,
        'integer'
    );

$query->select([
    'number_published' => $query->func()->count($publishedCase),
    'number_unpublished' => $query->func()->count($unpublishedCase)
]);
```

A função `addCase` também pode encadear várias instruções para criar `if .. then .. [elseif .. then ..] [. . else]` lógica dentro de seu SQL.

Se quisermos classificar as cidades em SMALL, MEDIUM ou LARGE, com base no tamanho da população, poderíamos fazer o seguinte:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->addCase(
            [
                $q->newExpr()->lt('population', 100000),
                $q->newExpr()->between('population', 100000, 999000),
                $q->newExpr()->gte('population', 999001),
            ],
            ['SMALL',  'MEDIUM', 'LARGE'], # valores que correspondem às condições
            ['string', 'string', 'string'] # tipo de cada valor
        );
    });
# WHERE CASE
#   WHEN population < 100000 THEN 'SMALL'
#   WHEN population BETWEEN 100000 AND 999000 THEN 'MEDIUM'
#   WHEN population >= 999001 THEN 'LARGE'
#   END
```

Sempre que houver menos condições de casos que valores, `addCase` produzirá automaticamente uma declaração `if .. then .. else`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
```

(continues on next page)

(continuação da página anterior)

```
        return $exp->addCase(
            [
                $q->newExpr()->eq('population', 0),
            ],
            ['DESERTED', 'INHABITED'], # values matching conditions
            ['string', 'string'] # type of each value
        );
    });
# WHERE CASE
#   WHEN population = 0 THEN 'DESERTED' ELSE 'INHABITED' END
```

Obtendo Matrizes em Vez de Entidades

Embora os ORMs e os conjuntos de resultados de objetos sejam poderosos, às vezes a criação de entidades é desnecessária. Por exemplo, ao acessar dados agregados, a construção de uma Entidade pode não fazer sentido. O processo de conversão dos resultados do banco de dados em entidades é chamado de hidratação. Se você deseja desativar esse processo, você pode fazer isso:

```
$query = $articles->find();
$query->enableHydration(false); // Resultados como matrizes em vez de entidades
$result = $query->toList(); // Execute a consulta e retorne a matriz
```

Depois de executar essas linhas, seu resultado deve ser semelhante a este:

```
[
    ['id' => 1, 'title' => 'First Article', 'body' => 'Article 1 body' ...],
    ['id' => 2, 'title' => 'Second Article', 'body' => 'Article 2 body' ...],
    ...
]
```

Adicionando Campos Calculados

Após suas consultas, talvez seja necessário fazer um pós-processamento. Se você precisar adicionar alguns campos calculados ou dados derivados, poderá usar o método `formatResults()`. Essa é uma maneira leve de mapear os conjuntos de resultados. Se você precisar de mais controle sobre o processo, ou desejar reduzir os resultados, use o recurso *Map/Reduce*. Se você estava consultando uma lista de pessoas, poderia calcular a idade delas com um formatador de resultados:

```
// Supondo que construímos os campos, condições e contenções.
$query->formatResults(function (\Cake\Collection\CollectionInterface $results) {
    return $results->map(function ($row) {
        $row['age'] = $row['birth_date']->diff(new \DateTime)->y;

        return $row;
    });
});
```

Como você pode ver no exemplo acima, a formatação de retornos de chamada receberá um `ResultSetDecorator` como seu primeiro argumento. O segundo argumento será a instância de consulta à qual o formatador foi anexado. O argumento `$results` pode ser percorrido e modificado conforme necessário.

Os formatadores de resultados são necessários para retornar um objeto iterador, que será usado como o valor de retorno para a consulta. As funções do formatador são aplicadas após a execução de todas as rotinas de Mapa/Redução. Os formatadores de resultados também podem ser aplicados a partir de associações contidas. O CakePHP garantirá que seus formatadores tenham um escopo adequado. Por exemplo, fazer o seguinte funcionaria conforme o esperado:

```
// Em um método na tabela Artigos
$query->contain(['Authors' => function ($q) {
    return $q->formatResults(function (\Cake\Collection\CollectionInterface $authors) {
        return $authors->map(function ($author) {
            $author['age'] = $author['birth_date']->diff(new \DateTime)->y;

            return $author;
        });
    });
}]);

// Obtem os resultados
$results = $query->all();

// Saída 29
echo $results->first()->author->age;
```

Como visto acima, os formatadores anexados aos criadores de consultas associados têm o escopo definido para operar apenas nos dados da associação. O CakePHP garantirá que os valores computados sejam inseridos na entidade correta.

Condições Avançadas

O construtor de consultas simplifica a criação de cláusulas complexas `where`. As condições agrupadas podem ser expressas fornecendo objetos `where()` e expressões. Para consultas simples, você pode criar condições usando uma matriz de condições:

```
$query = $articles->find()
    ->where([
        'author_id' => 3,
        'OR' => [['view_count' => 2], ['view_count' => 3]],
    ]);
```

O exemplo acima geraria SQL como:

```
SELECT * FROM articles WHERE author_id = 3 AND (view_count = 2 OR view_count = 3)
```

Se você preferir evitar matrizes profundamente aninhadas, use a chamada de retorno `where()` para criar suas consultas. O formulário de retorno de chamada permite que você use o construtor de expressões para criar condições mais complexas sem matrizes. Por exemplo:

```
$query = $articles->find()->where(function ($exp, $query) {
    // Use add() para adicionar várias condições para o mesmo campo.
    $author = $query->newExpr()->or_(['author_id' => 3])->add(['author_id' => 2]);
    $published = $query->newExpr()->and_(['published' => true, 'view_count' => 10]);

    return $exp->or_([
        'promoted' => true,
        $query->newExpr()->and_([$author, $published])
```

(continues on next page)

(continuação da página anterior)

```
    ]);
});
```

O exemplo acima irá gerar SQL semelhante a:

```
SELECT *
FROM articles
WHERE (
    (
        (author_id = 2 OR author_id = 3)
        AND
        (published = 1 AND view_count = 10)
    )
    OR promoted = 1
)
```

O objeto de expressão que é passado para as funções `where()` possui dois tipos de métodos. O primeiro tipo de método são **combinadores**. Os métodos `and_()` e `or_()` criam novos objetos de expressão que mudam **como** as condições são combinadas. O segundo tipo de métodos são **condições**. As condições são adicionadas a uma expressão em que são alinhadas com o combinador atual.

Por exemplo, chamar `$exp->and_(/* ... */)` criará um novo objeto `Expression` que combina todas as condições que ele contém com `AND`. Enquanto `$exp->or_()` criará um novo objeto `Expression` que combina todas as condições adicionadas a ele com `OR`. Um exemplo de adição de condições com um objeto `Expression` seria:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        return $exp
            ->eq('author_id', 2)
            ->eq('published', true)
            ->notEq('spam', true)
            ->gt('view_count', 10);
    });
```

Desde que começamos a usar `where()`, não precisamos chamar `and_()`, pois isso acontece implicitamente. A descrição acima mostra alguns métodos de condição novos combinados com `AND`. O SQL resultante seria semelhante:

```
SELECT *
FROM articles
WHERE (
author_id = 2
AND published = 1
AND spam != 1
AND view_count > 10)
```

Obsoleto desde a versão 3.5.0: A partir da versão 3.5.0, o método `orWhere()` está obsoleto. Este método é difícil prever o SQL com base no estado atual da consulta. Use `where()` para ter um comportamento mais previsível e mais fácil de entender

No entanto, se quisermos usar as condições `AND` e `OR`, poderíamos fazer o seguinte:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or_(['author_id' => 2])
```

(continues on next page)

(continuação da página anterior)

```
            ->eq('author_id', 5);

        return $exp
            ->add($orConditions)
            ->eq('published', true)
            ->gte('view_count', 10);
    });
```

O que geraria o SQL semelhante a:

```
SELECT *
FROM articles
WHERE (
(author_id = 2 OR author_id = 5)
AND published = 1
AND view_count >= 10)
```

Os métodos `or_()` e `and_()` também permitem usar funções como parâmetros. Muitas vezes, é mais fácil ler do que encadear métodos:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or_(function ($or) {
            return $or->eq('author_id', 2)
                ->eq('author_id', 5);
        });

        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

Você pode negar sub-expressões usando `not()`:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp) {
        $orConditions = $exp->or_(['author_id' => 2])
            ->eq('author_id', 5);

        return $exp
            ->not($orConditions)
            ->lte('view_count', 10);
    });
```

O que gerará o seguinte SQL:

```
SELECT *
FROM articles
WHERE (
NOT (author_id = 2 OR author_id = 5)
AND view_count <= 10)
```

Também é possível construir expressões usando as funções SQL:

```
$query = $articles->find()
    ->where(function (QueryExpression $exp, Query $q) {
        $year = $q->func()->year([
            'created' => 'identifier'
        ]);

        return $exp
            ->gte($year, 2014)
            ->eq('published', true);
    });
```

O que gerará o seguinte SQL:

```
SELECT *
FROM articles
WHERE (
YEAR(created) >= 2014
AND published = 1
)
```

Ao usar os objetos de expressão, você pode usar os seguintes métodos para criar condições:

- `eq()` Cria uma condição de igualdade:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->eq('population', '10000');
    });
# WHERE population = 10000
```

- `notEq()` Cria uma condição de desigualdade:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notEq('population', '10000');
    });
# WHERE population != 10000
```

- `like()` Cria uma condição usando o operador `LIKE`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->like('name', '%A%');
    });
# WHERE name LIKE "%A%"
```

- `notLike()` Cria uma condição `LIKE` negada:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notLike('name', '%A%');
    });
# WHERE name NOT LIKE "%A%"
```

- `in()` Cria uma condição usando `IN`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->in('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id IN ('AFG', 'USA', 'EST')
```

- `notIn()` Crie uma condição negada usando `IN`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->notIn('country_id', ['AFG', 'USA', 'EST']);
    });
# WHERE country_id NOT IN ('AFG', 'USA', 'EST')
```

- `gt()` Cria uma condição >:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->gt('population', '10000');
    });
# WHERE population > 10000
```

- `gte()` Cria uma condição >=:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->gte('population', '10000');
    });
# WHERE population >= 10000
```

- `lt()` Cria uma condição <:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->lt('population', '10000');
    });
# WHERE population < 10000
```

- `lte()` Cria uma condição <=:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->lte('population', '10000');
    });
# WHERE population <= 10000
```

- `isNull()` Cria uma condição `IS NULL`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->isNull('population');
    });
# WHERE (population) IS NULL
```

- `isNotNull()` Cria uma condição negada `IS NULL`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->isNotNull('population');
    });
# WHERE (population) IS NOT NULL
```

- `between()` Cria uma condição `BETWEEN`:

```
$query = $cities->find()
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->between('population', 999, 5000000);
    });
# WHERE population BETWEEN 999 AND 5000000,
```

- `exists()` Cria uma condição usando `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >' => 5000000]);

$query = $countries->find()
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->exists($subquery);
    });
# WHERE EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id AND␣
↪population > 5000000)
```

- `notExists()` Cria uma condição negada usando `EXISTS`:

```
$subquery = $cities->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('countries.id', 'cities.country_id');
    })
    ->andWhere(['population >' => 5000000]);

$query = $countries->find()
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->notExists($subquery);
    });
# WHERE NOT EXISTS (SELECT id FROM cities WHERE countries.id = cities.country_id␣
↪AND population > 5000000)
```

Em situações em que você não pode obter ou não deseja usar os métodos do construtor para criar as condições desejadas, também pode usar trechos de SQL nas cláusulas where:

```
// Compare dois campos entre si
$query->where(['Categories.parent_id != Parents.id']);
```

Aviso: Os nomes dos campos usados nas expressões e os snippets SQL nunca **devem** conter conteúdo não confiável. Veja a seção *Usando Funções SQL* para saber como incluir com segurança dados inseguros nas chamadas de função.

Usando Identificadores em Expressões

Quando você precisar fazer referência a uma coluna ou identificador SQL em suas consultas, poderá usar o método `identifier()`:

```
$query = $countries->find();
$query->select([
        'year' => $query->func()->year([$query->identifier('created')])
    ])
    ->where(function ($exp, $query) {
        return $exp->gt('population', 100000);
    });
```

Aviso: Para evitar injeções de SQL, as expressões Identifier nunca devem ter dados não confiáveis passados para elas.

Criando Cláusulas IN Automaticamente

Ao criar consultas usando o ORM, geralmente você não precisará indicar os tipos de dados das colunas com as quais está interagindo, pois o CakePHP pode inferir os tipos com base nos dados do esquema. Se em suas consultas você deseja que o CakePHP converta automaticamente a igualdade em comparações `IN`, será necessário indicar o tipo de dados da coluna:

```
$query = $articles->find()
    ->where(['id' => $ids], ['id' => 'integer[]']);

// Ou inclua IN para converter automaticamente em uma matriz.
$query = $articles->find()
    ->where(['id IN' => $ids]);
```

O exemplo acima criará automaticamente `id IN (...)` em vez de `id = ?`. Isso pode ser útil quando você não sabe se receberá um valor escalar ou matriz de parâmetros. O sufixo `[]` em qualquer nome de tipo de dados indica para o construtor de consultas que você deseja que os dados sejam tratados como uma matriz. Se os dados não forem uma matriz, eles serão convertidos em uma matriz. Depois disso, cada valor na matriz será convertido usando o *type system*. Isso funciona com tipos complexos também. Por exemplo, você pode pegar uma lista de objetos DateTime usando:

```
$query = $articles->find()
    ->where(['post_date' => $dates], ['post_date' => 'date[]']);
```

Criação Automática de IS NULL

Quando se espera que um valor de condição seja `null` ou qualquer outro valor, você pode usar o operador `IS` para criar automaticamente a expressão correta:

```
$query = $categories->find()
    ->where(['parent_id IS' => $parentId]);
```

O exemplo acima criará `parent_id` =: c1 ou `parent_id IS NULL`, dependendo do tipo de `$parentId`

Criação Automática de IS NOT NULL

Quando se espera que um valor de condição não seja `null` ou qualquer outro valor, você pode usar o operador `IS NOT` para criar automaticamente a expressão correta:

```
$query = $categories->find()
    ->where(['parent_id IS NOT' => $parentId]);
```

O exemplo acima criará `parent_id` != :c1 ou `parent_id IS NOT NULL`, dependendo do tipo de `$parentId`

Expressões Nativas

Quando você não pode construir o SQL necessário usando o construtor de consultas, pode usar objetos de expressão para adicionar trechos de SQL às suas consultas:

```
$query = $articles->find();
$expr = $query->newExpr()->add('1 + 1');
$query->select(['two' => $expr]);
```

`Expression` objetos podem ser usados com qualquer método do construtor de consultas, como `where()`, `limit()`, `group()`, `select()` e muitos outros métodos.

Aviso: O uso de objetos de expressão deixa você vulnerável à injeção de SQL. Você nunca deve usar dados não confiáveis em expressões.

Obtendo Resultados

Depois de fazer sua consulta, você precisará recuperar linhas dela. Existem algumas maneiras de fazer isso:

```
// Iterar a consulta
foreach ($query as $row) {
    // Fazer algumas coisas.
}

// Obtêm os resultados
$results = $query->all();
```

Você pode usar *qualquer um dos métodos de coleção* nos objetos de consulta para pré-processar ou transformar os resultados:

```
// Use um dos métodos de coleção.
$ids = $query->map(function ($row) {
    return $row->id;
});

$maxAge = $query->max(function ($max) {
    return $max->age;
});
```

Você pode usar `first` ou `firstOrFail` para recuperar um único registro. Esses métodos alterarão a consulta adicionando uma cláusula `LIMIT 1`:

```
// Obtenha apenas a primeira linha
$row = $query->first();

// Obtenha a primeira linha ou uma exceção.
$row = $query->firstOrFail();
```

Retornando a Contagem Total de Registros

Usando um único objeto de consulta, é possível obter o número total de linhas encontradas para um conjunto de condições:

```
$total = $articles->find()->where(['is_active' => true])->count();
```

O método `count()` ignorará as cláusulas `limit`, `offset` e `page`, portanto, o seguinte retornará o mesmo resultado:

```
$total = $articles->find()->where(['is_active' => true])->limit(10)->count();
```

Isso é útil quando você precisa conhecer o tamanho total do conjunto de resultados com antecedência, sem precisar construir outro objeto `Query`. Da mesma forma, todas as rotinas de formatação e redução de mapa são ignoradas ao usar o método `count()`.

Além disso, é possível retornar a contagem total de uma consulta contendo cláusulas de grupo sem precisar reescrever a consulta de nenhuma maneira. Por exemplo, considere esta consulta para recuperar IDs de artigos e contagem de seus comentários:

```
$query = $articles->find();
$query->select(['Articles.id', $query->func()->count('Comments.id')])
    ->matching('Comments')
    ->group(['Articles.id']);
$total = $query->count();
```

Após a contagem, a consulta ainda pode ser usada para buscar os registros associados:

```
$list = $query->all();
```

Às vezes, convém fornecer um método alternativo para contar o total de registros de uma consulta. Um caso de uso comum para isso é fornecer um valor em cache ou uma estimativa do total de linhas ou alterar a consulta para remover partes desnecessariamente caras, como left joins. Isso se torna particularmente útil ao usar o sistema de paginação do CakePHP que chama o método `count()`:

```
$query = $query->where(['is_active' => true])->counter(function ($query) {
    return 100000;
});
$query->count(); // Retorna 100000
```

No exemplo acima, quando o componente de paginação chamar o método count, ele receberá o número estimado de linhas codificadas

Cache de Resultados Carregados

Ao buscar entidades que não mudam com frequência, convém armazenar em cache os resultados. A classe `Query` torna isso simples:

```
$query->cache('recent_articles');
```

Ativará o cache no conjunto de resultados da consulta. Se apenas um argumento for fornecido para `cache()`, a configuração de cache "padrão" será usada. Você pode controlar qual configuração de armazenamento em cache é usada com o segundo parâmetro:

```
// Nome da configuração.
$query->cache('recent_articles', 'dbResults');

// Instância de CacheEngine
$query->cache('recent_articles', $memcache);
```

Além de suportar chaves estáticas, o método `cache()` aceita uma função para gerar a chave. A função que você fornecer receberá a consulta como argumento. Você pode ler aspectos da consulta para gerar dinamicamente a chave de cache:

```
// Gere uma chave com base em uma soma de verificação simples
// da cláusula where da consulta
$query->cache(function ($q) {
    return 'articles-' . md5(serialize($q->clause('where')));
});
```

O método de cache simplifica a adição de resultados em cache aos seus finders personalizados ou através dos ouvintes de eventos.

Quando os resultados de uma consulta em cache são buscados, acontece o seguinte:

1. Se a consulta tiver resultados definidos, eles serão retornados.
2. A chave do cache será resolvida e os dados do cache serão lidos. Se os dados do cache não estiverem vazios, esses resultados serão retornados.
3. Se o cache falhar, a consulta será executada, o evento `Model.beforeFind` será acionado e um novo `ResultSet` será criado. Este `ResultSet` será gravado no cache e retornado.

Nota: Você não pode armazenar em cache um resultado de consulta de streaming.

Carregando Associações

O construtor pode ajudá-lo a recuperar dados de várias tabelas ao mesmo tempo com a quantidade mínima de consultas possível. Para poder buscar dados associados, primeiro você precisa configurar associações entre as tabelas, conforme descrito na seção *Associações - Conectando tabelas*. Essa técnica de combinar consultas para buscar dados associados de outras tabelas é chamada **carregamento rápido**.

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);
    }]);
```

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']
    ]
]);
```

Filtrando por Dados Aassociados

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) {
```

(continues on next page)

(continuação da página anterior)

```
    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.

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.

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.

Adicionando Junções

Além de carregar dados relacionados com `contains()`, você também pode adicionar junções adicionais com o construtor de consultas:

```
$query = $articles->find()
    ->join([
        'table' => 'comments',
        'alias' => 'c',
        'type' => 'LEFT',
        'conditions' => 'c.article_id = articles.id',
    ]);
```

Você pode anexar várias junções ao mesmo tempo passando uma matriz associativa com várias junções:

```
$query = $articles->find()
    ->join([
```

(continues on next page)

(continuação da página anterior)

```
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => 'c.article_id = articles.id',
        ],
        'u' => [
            'table' => 'users',
            'type' => 'INNER',
            'conditions' => 'u.id = articles.user_id',
        ]
    ]);
```

Como visto acima, ao adicionar junções, o alias pode ser a chave da matriz externa. As condições de junção também podem ser expressas como uma matriz de condições:

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.created >' => new DateTime('-5 days'),
                'c.moderated' => true,
                'c.article_id = articles.id'
            ]
        ],
    ], ['c.created' => 'datetime', 'c.moderated' => 'boolean']);
```

Ao criar junções manualmente e usar condições baseadas em matriz, é necessário fornecer os tipos de dados para cada coluna nas condições de junção. Ao fornecer tipos de dados para as condições de junção, o ORM pode converter corretamente os tipos de dados em SQL. Além de `join()`, você pode usar `rightJoin()`, `leftJoin()` e `innerJoin()` para criar junções:

```
// Join com um alias e condições de string
$query = $articles->find();
$query->leftJoin(
    ['Authors' => 'authors'],
    ['Authors.id = Articles.author_id']);

// Join com um alias, matriz de condições e tipos
$query = $articles->find();
$query->innerJoin(
    ['Authors' => 'authors'],
    [
    'Authors.promoted' => true,
    'Authors.created' => new DateTime('-5 days'),
    'Authors.id = Articles.author_id'
    ],
    ['Authors.promoted' => 'boolean', 'Authors.created' => 'datetime']);
```

Deve-se observar que, se você definir a opção `quoteIdentifiers` como `true` ao definir sua `Conexão`, as condições de junção entre os campos da tabela deverão ser definidas da seguinte forma:

```
$query = $articles->find()
    ->join([
        'c' => [
            'table' => 'comments',
            'type' => 'LEFT',
            'conditions' => [
                'c.article_id' => new \Cake\Database\Expression\IdentifierExpression(
↪'articles.id')
            ]
        ],
    ]);
```

Isso garante que todos os seus identificadores sejam citados em toda a consulta, evitando erros com alguns drivers de banco de dados (notavelmente no PostgreSQL)

Inserindo Dados

Diferente dos exemplos anteriores, você não deve usar `find()` para criar consultas de inserção. Em vez disso, crie um novo objeto `Query` usando `query()`:

```
$query = $articles->query();
$query->insert(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text'
    ])
    ->execute();
```

Para inserir várias linhas com apenas uma consulta, você pode encadear o método `values()` quantas vezes for necessário:

```
$query = $articles->query();
$query->insert(['title', 'body'])
    ->values([
        'title' => 'First post',
        'body' => 'Some body text'
    ])
    ->values([
        'title' => 'Second post',
        'body' => 'Another body text'
    ])
    ->execute();
```

Geralmente, é mais fácil inserir dados usando entidades e *save()*. Ao compor uma consulta `SELECT` e `INSERT` juntas, você pode criar consultas de estilo `INSERT INTO ... SELECT`

$select = $articles->find()
->select([“title”, “body”, “published”]) ->where([“id” => 3]);

$query = $articles->query()
->insert([“title”, “body”, “published”]) ->values($select) ->execute();

Nota: A inserção de registros com o construtor de consultas não acionará eventos como `Model.afterSave`. Em vez

disso, você deve usar o *ORM para salvar dados*.

Atualizando Dados

Como nas consultas de inserção, você não deve usar `find()` para criar consultas de atualização. Em vez disso, crie um novo objeto `Query` usando `query()`:

```
$query = $articles->query();
$query->update()
    ->set(['published' => true])
    ->where(['id' => $id])
    ->execute();
```

Geralmente, é mais fácil atualizar dados usando entidades e `patchEntity()`.

Nota: A atualização de registros com o construtor de consultas não acionará eventos como `` Model.afterSave``. Em vez disso, você deve usar o *ORM para salvar os dados*.

Apagando Dados

Como nas consultas de inserção, você não deve usar `find()` para criar consultas de exclusão. Em vez disso, crie um novo objeto de consulta usando `query()`:

```
$query = $articles->query();
$query->delete()
    ->where(['id' => $id])
    ->execute();
```

Generally, it is easier to delete data using entities and *delete()*.

Geralmente, é mais fácil excluir dados usando entidades e *delete()*.

Prevenção de SQL Injection

Embora as camadas de abstração do ORM e do banco de dados evitem a maioria dos problemas de injeção de SQL, ainda é possível deixar-se vulnerável por uso inadequado.

Ao usar matrizes de condições, a chave/lado esquerdo e as entradas de valor único não devem conter dados do usuário:

```
$query->where([
    // Os dados no lado esquerdo/chave não são seguros, pois serão
    // inserido na consulta gerada como está
    $userData => $value,

    // O mesmo se aplica às entradas de valor único, elas não são
        // seguras para usar com os dados do usuário de qualquer forma
    $userData,
    "MATCH (comment) AGAINST ($userData)",
    'created < NOW() - ' . $userData
]);
```

Ao usar o construtor de expressões, os nomes das colunas não devem conter dados do usuário:

```
$query->where(function (QueryExpression $exp) use ($userData, $values) {
    // Os nomes de colunas em todas as expressões não são seguras.
    return $exp->in($userData, $values);
});
```

Ao criar expressões de função, os nomes de funções nunca devem conter dados do usuário:

```
// Não é seguro.
$query->func()->{$userData}($arg1);

// Também não é seguro usar uma matriz de
// dados do usuário em uma expressão de função
$query->func()->coalesce($userData);
```

Expressões brutas nunca são seguras:

```
$expr = $query->newExpr()->add($userData);
$query->select(['two' => $expr]);
```

Valores de Ligação

É possível proteger contra muitas situações inseguras usando ligações. Semelhante a *vinculando valores a instruções preparadas*, os valores podem ser vinculados a consultas usando o método `Cake\Database\Query::bind()`

O exemplo a seguir seria uma variante segura do exemplo inseguro, propenso a injeção de SQL, dado acima:

```
$query
    ->where([
        'MATCH (comment) AGAINST (:userData)',
        'created < NOW() - :moreUserData'
    ])
    ->bind(':userData', $userData, 'string')
    ->bind(':moreUserData', $moreUserData, 'datetime');
```

Nota: Ao contrário de `Cake\Database\StatementInterface::bindValue()`, `Query::bind()` requer passar os espaços reservados nomeados, incluindo os dois pontos!

Mais Consultas Complexas

O construtor de consultas é capaz de criar consultas complexas, como consultas e subconsultas `UNION`.

Unions

As Unions são criadas compondo uma ou mais consultas selecionadas juntas:

```
$inReview = $articles->find()
    ->where(['need_review' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->union($inReview);
```

Você pode criar consultas `UNION ALL` usando o método `unionAll()`:

```
$inReview = $articles->find()
    ->where(['need_review' => true]);

$unpublished = $articles->find()
    ->where(['published' => false]);

$unpublished->unionAll($inReview);
```

Subconsultas

As subconsultas são um recurso poderoso nos bancos de dados relacionais e sua criação no CakePHP é bastante intuitiva. Ao compor consultas em conjunto, você pode criar subconsultas:

```
// Antes da versão 3.6.0, use o association().
$matchingComment = $articles->getAssociation('Comments')->find()
    ->select(['article_id'])
    ->distinct()
    ->where(['comment LIKE' => '%CakePHP%']);

$query = $articles->find()
    ->where(['id IN' => $matchingComment]);
```

Subqueries are accepted anywhere a query expression can be used. For example, in the `select()` and `join()` methods.

Subconsultas são aceitas em qualquer lugar em que uma expressão de consulta possa ser usada. Por exemplo, nos métodos `select()` e `join()`

Adicionando Instruções de Bloqueio

A maioria dos fornecedores de bancos de dados relacionais suporta a remoção de bloqueios ao executar operações selecionadas. Você pode usar o método `epilog()` para este:

```
// Em MySQL
$query->epilog('FOR UPDATE');
```

O método `epilog()` permite anexar SQL bruto ao final das consultas. Você nunca deve colocar dados brutos do usuário em `epilog()`

Executando Consultas Complexas

Embora o construtor de consultas facilite a criação da maioria das consultas, consultas muito complexas podem ser entediantes e complicadas. Você pode *executar o SQL desejado diretamente*.

A execução direta do SQL permite ajustar a consulta que será executada. No entanto, isso não permite que você use `contains` ou outros recursos ORM de nível superior.

Objetos de tabela

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.

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 *sintaxe plugin* e fornecer opções de configuração adicionais:

```
namespace App\Model\Table;

use Cake\ORM\Table;
```

(continues on next page)

[87] https://github.com/cakephp/docs

(continuação da página anterior)

```
class ArticlesTable extends Table
{
    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 convensão, classes de entidades ficam em **src/Model/Entity/**. Se a nossa aplicação tem um tabela `articles`, poderiamos 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 entitdade 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')
]);
```

Outro maneira de obter novas entidades é usando o método `newEntity()` dos objetos `Table`:

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$article = TableRegistry::getTableLocator()->get('Articles')->newEntity();

$article = TableRegistry::getTableLocator()->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 `_fields` 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 atribuidas 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 loos, já que o CakePHP não impedirá os métodos mutadores de looping infinitos.

Os mutadores permitem você converter as propriedades conforme são atribuidas, 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)
```

(continues on next page)

(continuação da página anterior)

```
    {
        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->_fields['first_name'] . '  ' .
            $this->_fields['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 proprieades 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();
```

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 errors
$errors = $user->getErrors();
// Antes da versão 3.4.0
$errors = $user->errors();

// Obtem os errors 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 array $_accessible = [
        'title' => true,
        'body' => true
    ];
}
```

Além dos campos concretos, existe um campo especial * que define o comportamento de falbback se um campo não for especificamente nomeado:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_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 Vigiados em Tempo de Execução

Você pode modificar a lista de campos vigiados 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 entnidade 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 exister momentos em que você precisa carregar seus dados sobre demanda (lazy load). Antes de entrar em como carregar associaçes 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 posso 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[88]

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;
}
```

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 convensão 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
{
```

(continues on next page)

[88] https://github.com/jeremyharris/cakephp-lazyload

(continuação da página anterior)

```
    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 signinifica 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 torna-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**

Enquanto os objetos Table fornecem uma abstração em torno de um «repositório» ou coleção de objetos, quando você consulta registros individuais, obtém objetos Entity. 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 pela chave primária.
$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 metódo `find()` usa um cache integrado. Você pode usar 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 customizavél *Personalizando Metódos 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, precisará carregá-las e gostará muito de fazer isso. O caminho mais simples para carregar uma Entidade ou objetos relacionais metódo `find()`. find provê um extensivél e facíl caminho para procurar e retornar dados, talves você se interesse por in:

```
// No controller ou table.

// Procure todos os artigos
$query = $articles->find('all');
```

O valor retornado por qualquer metódo `find()` será sempre um *Cake\ORM\Query* objeto. A class Query assim permitindo que possa posteriormente refinar a consulta depois de cria lá. Objeto Query não será executado até que inicie um busca por linhas, seja convertido num array, ou chamado outro metódo, 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 específica, limites, incluíndo associação ou uma interface mais fluênte, ou busca de resultados por id de usuário lógado.

```
// 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 metódos simulados:

```
// No controller ou table.
$query = $articles->find('all', [
    'conditions' => ['Articles.created >' => new DateTime('-10 days')],
```

(continues on next page)

(continuação da página anterior)

```
    '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 funçoes agregadas.
- `having` adicionar HVAING.
- `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 metódo `getOptions()` no objeto para retornar as opções utilizadas. Quando uma consulta for passada para o controller, recomendamos uma leitura sobre consultas personalizadas em *Personalizando Metódos de Consulta*. Usando metódos 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 metódo `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 metódo `get()` caso queira carregar uma entidade pelo chave primária.

Nota: O metódo `first()` retorna `null` caso nenhum resultado seja encontrado.

Contando os resultados

Criando uma consulta você gosta do metódo `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 *Retornando a Contagem Total de Registros* para modos de uso diferentes com o metódo `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 metódo 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',
```

(continues on next page)

(continuação da página anterior)

```
    '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 idenficação 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 metódos criados em nossas entidades, isto também é possível no metódo "list". . Neste exemplo mostra o uso metódo 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 metódo mutador criado na entidade
//Onde ao juntar o 1 nome com o 2 formará o nome completo.
```

Encontrando dados enfileirados

O metódo `find('threaded')` retorna que estarão relacionados por chaves. Por padrão o Cake usa o campo chave `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 possue 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 *Árvore* posteriormente.

Personalizando Metódos 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 metódo personalizado de busca pode ser ideal para simplificar processos, consultar dados complexos, otimizar buscas, ou criar uma busca padrão em um metódo simplificado feito por você. Eles podem ser definidos na criação do objeto tabela e devem obedecer a conveção padrão do Cake. Ao criar um metódo 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 metódo personalizado
    public function findOwnedBy(Query $query, array $options)
    {
        $user = $options['user'];
```

(continues on next page)

(continuação da página anterior)

```
        return $query->where(['author_id' => $user->id]);
    }

}

// No controller ou table.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find('ownedBy', ['user' => $userEntity]);
//Retorne todos os artigos, quero que seja de meu usuário, porém somente os já␣
↪publicados.
```

O metódo 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 irelevante. 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.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 metódos de busca, onde na chamada do metódo poderá apenas adicionar o nome do campo que desejar 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

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->get('Users');
// Duas chamadas também iguais.
$query = $users->findByUsername('joebob');
$query = $users->findAllByUsername('joebob');
```

Pode usar também multiplos 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 metódo. Também não será possível associar dados com o atributo `contain`, pois não é compatível com buscas dinâmicas. Lembra-se dos nossos queridos *Personalizando Metódos 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 metódos de busca podem ser simples, porém eles trazem uma sobrecargar adicional, pelo fato de ser necessário enteder 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 relacinados, 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 Via Matching And Joins*.

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
```

(continues on next page)

(continuação da página anterior)

```
            ->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);
    }]);
```

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 Via Matching And Joins

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.

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.

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.

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.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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

// Prior to 3.6 use TableRegistry::get('Articles')
```

(continues on next page)

(continuação da página anterior)

```
$articles = TableRegistry::getTableLocator()->get('Articles');
$results = $articles->find()->contain(['Authors'])->all();

$authorList = $results->combine('id', 'author.name');
```

The *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 => $articles) {
    echo sprintf("The are %d %s articles", count($articles), $status);
}
```

The above will ouput 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 (stripos('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']);
```

(continues on next page)

(continuação da página anterior)

```
    $mr->emitIntermediate($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 Metódos 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, sinta-se a vontade para nos enviar um *pull request* para o Github[89] 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, atravéz 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;

// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::getTableLocator()->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;
}
```

[89] https://github.com/cakephp/docs

Atualizando Dados

Atualizar seus dados é igualmente fácil, e o método `save()` também é usado para esse propósito:

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::getTableLocator()->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:

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articlesTable = TableRegistry::getTableLocator()->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 demostrarão 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

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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' => []]);
```

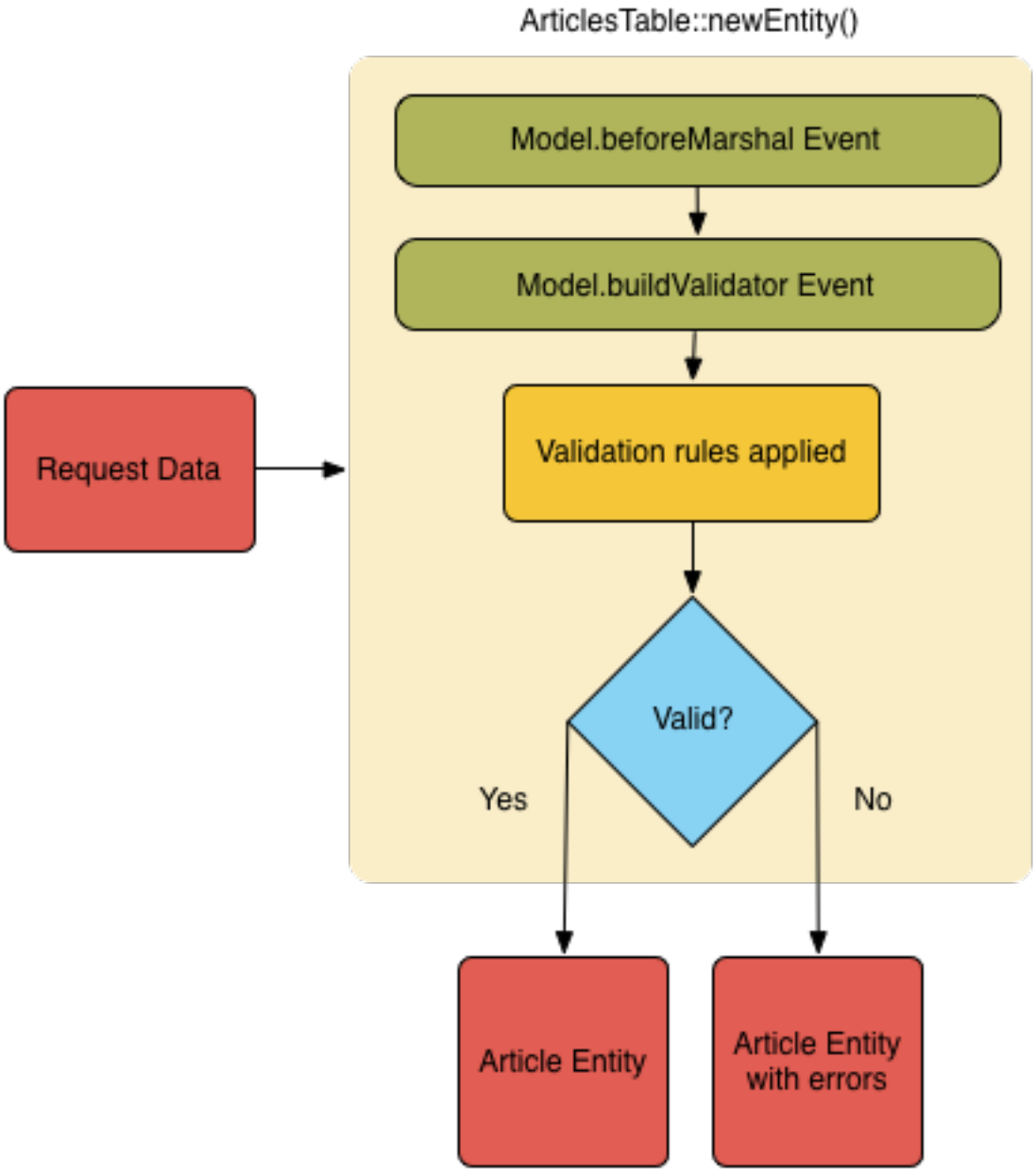
Os dados associados também são validados por padrão, a menos que seja informado o contrário. Você também pode alterar o conjunto de validação a ser usada por associação:

```
// No controller

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');

// Pular validação na associação de Tags e
// Definino 'signup' como método de validação para Comments.Users
$entity = $articles->newEntity($this->request->getData(), [
    'associated' => [
        'Tags' => ['validate' => false],
        'Comments.Users' => ['validate' => 'signup']
    ]
]);
```

O capitulo *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 conterá erros, e quaisquer campos inválidos não serão preenchidos na entidade 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.

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 associaçoes 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.

Convertendo Vários Registros

Ao criar formulários que cria/atualiza vários registros ao mesmo tempo, você pode usar o método `newEntities()`:

```
// No controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 acessiveis. 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

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 o opção `validate` assim:

```
// No controller.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 array $_accessible = [
    // .. outras propriedades
   'tags' => true,
];
```

Nota: Para as associações hasMany e belongsToMany, se houvesse algumas entidades que 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);
```

(continues on next page)

(continuação da página anterior)

```
$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.

// Prior to 3.6 use TableRegistry::get('Comments')
$comments = TableRegistry::getTableLocator()->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.

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$list = $articles->find('popular')->toArray();
$patched = $articles->patchEntities($list, $this->request->getData());
```

(continues on next page)

(continuação da página anterior)

```
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 ínicio 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)) {
```

(continues on next page)

(continuação da página anterior)

```
            $data[$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 consitentes.

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:

```
// Contêm ['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 atribuidas para associações:

```
// Apenas permite alterar o title e tags
// e nome da tag é a única columa 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 funcões 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

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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:

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 trasação 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:

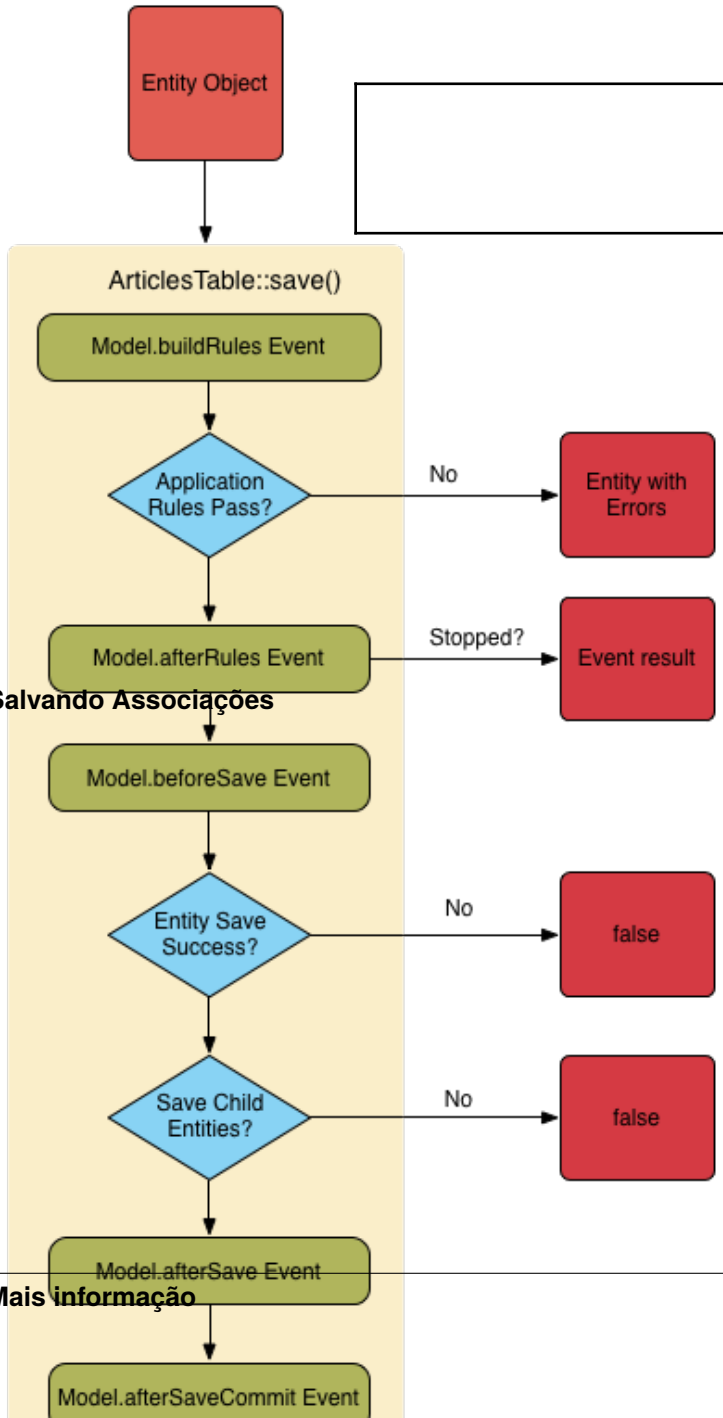
```
INSERT INTO articles (title) VALUES ('My new title');
```

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 pais 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:



Consule a seção *Aplicando regras da aplicação* para mais informação sobre como criar e usar regras.

Aviso: Se nenhuma alter
tidade quando ela é salva
serão disparado porque o s
tado.

O método `save()` retornará a entidade modificada quando sucesso, e `false` quando falhar. Você pode desativar regras e/ou transações usando o argumento `$options` para salvar:

```
// Num␣
↪método de controller ou model
$articles-
↪>save($article, ['checkRules
↪' => false, 'atomic' => false]);
```

Salvando Associações

Quando você está salvando uma entidade, você também pode escolher de salvar alguma ou todas as entidades associadas. Por padrão, todos as entidades de primeiro nível serão salvas. Por exemplo salvando um Artigo, você também atualizará todas as entidades modificadas (dirty) que são diretamente realicionadas a tabela de artigos.

Você pode ajustar as associações que são salvas usando a opção `associated`:

```
// Num controller.
```

(continues on next page)

Mais informação

(continuação da página anterior)

```
// 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␣
↪(endereços) 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'
    ]
];
```

(continues on next page)

(continuação da página anterior)

```
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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'
    ]
];

// Prior to 3.6 use TableRegistry::get('Users')
$users = TableRegistry::getTableLocator()->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.']
    ]
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 estrategias de salvamento:

append
: Os registros associados são atualizados no banco de dados ou, se não econtrado 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`. Consosule *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']
    ]
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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 checkox 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 estrategias 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 relevente 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 associaçao, 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 propriendade 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 apartir 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\TypeFactory;
TypeFactory::map('json', 'Cake\Database\Type\JsonType');

// No src/Model/Table/UsersTable.php
use Cake\Database\Schema\TableSchemaInterface;

class UsersTable extends Table
{

    public function getSchema(): TableSchemaInterface
    {
        $schema = parent::getSchema();
        $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\ORMException\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.

Salvando Várias Entidades

`Cake\ORM\Table::saveMany`(*$entities*, *$options = []*)

Usando esse método você pode salvar várias entidades atomicamente. `$entities` podem ser um array de entidades criadas usando `newEntities()` / `patchEntities()`. `$options` pode ter as mesmas opções aceitas por `save()`:

```
$data = [
    [
        'title' => 'First post',
```

(continues on next page)

(continuação da página anterior)

```
        'published' => 1
    ],
    [
        'title' => 'Second post',
        'published' => 1
    ],
];

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->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.

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 considera 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 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 "cascate" 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 masa. 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\ORMException\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.

Associações - Conectando tabelas

Definindo a relação entre diferentes objetos na sua aplicação deveria ser um processo natural. Por exemplo, um artigo deve ter muitos comentários, e pertencer a um autor. Autores deve ter muitos artigos e logo muitos comentários. O CakePHP torna fácil a gestão das relações entre os modelos. As quatro tipo de associações no CakePHP são: hasOne, hasMany, belongsTo, and belongsToMany.

Relação	Tipo de associação	Exemplo
um para um	hasOne	A user has one profile.
um para muitos	hasMany	A user can have multiple articles.
muitos para um	belongsTo	Many articles belong to a user.
muitos para muitos	belongsToMany	Tags belong to many articles.

Associações são definidas durante o método `initialize()` do objeto da sua tabela. Métodos que fechem com o tipo de associação permitem a você definir as associações da sua aplicação. Por exemplo se nós quisermos definir uma associação do tipo belongsTo em nosso ArticlesTable:

```
namespace App\Model\Table;

use Cake\ORM\Table;

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

A forma mais simples de definição de qualquer associação é usar o alias do modelo que você quer associar. Por padrão todos os detalhes das associações serão usadas as convenções do CakePHP. Se você quiser customizar a forma como as suas associações são trabalhadas, você pode modificar elas com os setters:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsTo('Authors', [
                'className' => 'Publishing.Authors'
            ])
            ->setForeignKey('authorid')
            ->setProperty('person');
    }
}
```

Você pode também usar arrays para customizar suas associações:

```
$this->belongsTo('Authors', [
    'className' => 'Publishing.Authors',
```

(continues on next page)

(continuação da página anterior)

```
    'foreignKey' => 'authorid',
    'propertyName' => 'person'
]);
```

Arrays, contudo, não oferecem o autocompletar, que uma IDE proporciona.

A mesma tabela pode ser usada múltiplas vezes para definir diferentes tipos de associações. Por exemplo considere um caso onde você deseja separar os comentários aprovados daqueles que ainda não foram moderados ainda:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasMany('Comments')
            ->setConditions(['approved' => true]);

        $this->hasMany('UnapprovedComments', [
                'className' => 'Comments'
            ])
            ->setConditions(['approved' => false])
            ->setProperty('unapproved_comments');
    }
}
```

Como você pode ver, por especificar a chave da `className` , é possível utilizar a mesma tabela em diferentes associações para a mesma tabela. Você até mesmo pode criar uma auto associação da tabela, criando uma estrutura de relação pai-filho:

```
class CategoriesTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasMany('SubCategories', [
            'className' => 'Categories'
        ]);

        $this->belongsTo('ParentCategories', [
            'className' => 'Categories'
        ]);
    }
}
```

Você também pode definir associações em massa ao criar uma única chamada, para `Table::addAssociations()` onde aceita um array contendo o nome das tabelas por associação como um argumento:

```
class PostsTable extends Table
{
    public function initialize(array $config)
    {
       $this->addAssociations([
           'belongsTo' => [
               'Users' => ['className' => 'App\Model\Table\UsersTable']
           ],
```

(continues on next page)

(continuação da página anterior)

```
            'hasMany' => ['Comments'],
            'belongsToMany' => ['Tags']
        ]);
    }
}
```

Cada associação aceita multiplas associações onde a chave é o alias, e a o valor são os dados da configuração da associação. Se a chave for usada for numérica, os valores serão tratados como aliases de associações.

Associação HasOne

Vamos definir uma tabela Users com uma relação hasOne para o endereço da tabela.

Primeiramente, a sua tabela do seu banco de dados precisa de uma chave correta. Para uma relação hasOne funcionar, uma tabela precisa conter uma chave estrangeira que aponte para um registro em otura tabela. Neste caso o endereço da tabela vai conter um campo chamado `user_id`. O padrão básico é:

hasOne: o *outro* modelo contendo a chave estrangeira.

Relação	Schema
Users hasOne Addresses	addresses.user_id
Doctors hasOne Mentors	mentors.doctor_id

Nota: Não necessariamente precisa seguir as convenções do CakePHP, você pode sobrescrever o uso de qualquer chave estrangeira na definição das suas associações. Mesmo assim ao aderir as convenções o seu código se torna menos repetitivo, logo, mais fácil de ler e de manter.

Se nos tivermos as classes `UsersTable` e `AddressesTable` fará com que possamos criar as associações da seguinte forma:

```
class UsersTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasOne('Addresses');
    }
}
```

Se você necessitar mais controle, você pode definir as suas associações usando os setters. Por exemplo, você deseja limitar as associações para incluir somente certos registros:

```
class UsersTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasOne('Addresses')
            ->setName('Addresses')
            ->setConditions(['Addresses.primary' => '1'])
            ->setDependent(true);
```

(continues on next page)

(continuação da página anterior)

```
    }
}
```

As chaves possíveis para uma relação hasOne incluem:

- **className**: o nome da classe que está sendo associada com o modelo atual. Se você está definindo uma relação "User hasOne Address", o nome da caheve da classe deve ser igual a "Addresses".
- **foreignKey**: o nome da chave estrangeira na outra tabela. Este é especialmente acessivel se você precisa definir vários relacionamentos hasOne. O valor padrão para esta chave é o nome sublinhado, singular do modelo atual, seguido do sufixo com "_id". No exemplo acima, seria o padrão para "user_id".
- **bindingKey**: O nome da coluna na tabela atual, que será usado para combinar com``foreignKey``. Se não for especificado, a chave primária (para exemplo, a coluna de identificação da tabela `Users`) será usado.
- **conditions**: um array de find() compatível com as condições como `['Addresses.primary' => true]`
- **joinType**: o tipo do join a ser usado para o SQL query, o padão é LEFT. Você pode usar o INNER se a sua associação é hasOne e estiver sempre presente.
- **dependent**: Quando a chave dependente é definida como `true`, e uma entidade é deletada, os registros do modelo associado tabém devem ser excluídos. Neste caso nós precisamos definir isto para `true` então ao deletar um usuário fará excluir todos os resgistros associados àquele registro.
- **cascadeCallbacks**: Quando este e o **dependent** são `true`, o deletar em cascata vai carregar e excluir todos os registros. Quando `false`, `deleteAll()` deve ser usado para remover o dados associados e não haverá callback disparado.
- **propertyName**: O nome da propriedade deve ser preenchido com os dados de uma tabela associada para a tabela fonte. Por default este é o nome sublinhado e singular das associações para `address` no nosso exemplo.
- **strategy**: Definindo a estratégia de query a ser utilizada . Por padrão o "join". O outro valor válido é "select", aos quais utiliza uma query separada.
- **finder**: O método finder a ser usado quando carregamos os registros associados.

Uma vez que as associações foram definidas, as operações de find na tabela de usuarios podem conter os registros de endereços se estes existirem:

```
// No controlelr ou no método da table.
$query = $users->find('all')->contain(['Addresses']);
foreach ($query as $user) {
    echo $user->address->street;
}
```

Abaixo emitirá um SQL similar a:

```
SELECT * FROM users INNER JOIN addresses ON addresses.user_id = users.id;
```

Associações BelongsTo

Agora que nós temos acesso aos registros de endereço através da tabela usuários, vamos criar a associação belongsTo em endereços a fim de ter acesso aos registros relacionados a usuário. A associação belongsTo é um complemento natural a associações do tipo hasOne e hasMany - isto permite a nós vermos dados relacionados a partir da outra perspectiva.

Quando definindo as chaves do seu banco de dados para uma relação pertence a (belongsTo) na sua tabela siga estas convenções:

belongsTo: ao *atual* modelo contendo a chave estrangeira.

Relação	Schema
Addresses belongsTo Users	addresses.user_id
Mentors belongsTo Doctors	mentors.doctor_id

Dica: Se a tabela contem uma chave estrangeira, isto pertence a outra tabela.

Nós podemos definir uma relação belongsTo em nosso Addresses table como:

```
class AddressesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsTo('Users');
    }
}
```

Nós também podemos definir uma relação mais especifica usando os setters:

```
class AddressesTable extends Table
{
    public function initialize(array $config)
    {
        // Prior to 3.4 version, use foreignKey() and joinType()
        $this->belongsTo('Users')
            ->setForeignKey('user_id')
            ->setJoinType('INNER');
    }
}
```

Chaves possiveis para associações belongsTo arrays inclui:

- **className**: o nome da classe do modelo que está sendo associado ao modelo atual. Se você está definindo que a relação "Profile belongsTo User", a chave da className deveria ser igual a "Users".
- **foreignKey**: o nome da chave estrangeira encontrada na tabela atual. Este é especialmente acessível, se você precisar definir multiplas relações belongsTo ao mesmo modelo. O valor padrão para esta chave é o sublinhado, e nome singular do outro modelo, com o sufixo `_id`.

- **bindingKey**: O nome da coluna da outra tabela, este ser´a usado para o match de `foreignKey`. Se não especificado, a chave primária (por exemplo o id da tabela `Users`) será usado.
- **conditions**: como um array para o find(), compativel com as condições ou SQL como `['Users.active' => true]`
- **joinType**: o tipo de join a ser usado na query SQL, por padrão é LEFT aos quais não deve atender as suas necessidades para todas as situações, INNER deve ajudar quando você quiser do seu modelo ou associados ou nenhum deles.
- **propertyName**: A propriedade nome deve preenchida com os dados das tabelas associadas e dos resultados. Pos padrão este é o sublinhado e nome singular da associação, então `user` em nosso exemplo.
- **strategy**: Define uma estratégia de query a ser usada. Por padrão o "join". O outro valor valido é "select", aos quais utiliza uma query separada.
- **finder**: O método finder é usado quando carregamos registros associados.

Uma vez que esta associação é definida, operações do tipo find para a tabela Addresses pode conter o registro de User se este existir:

```
// No controller ou no metodo de table.
$query = $addresses->find('all')->contain(['Users']);
foreach ($query as $address) {
    echo $address->user->username;
}
```

Abaixo emite um SQL que é similar a:

```
SELECT * FROM addresses LEFT JOIN users ON addresses.user_id = users.id;
```

Associações HasMany

Um exemplo de associações hasMany é «Article hasMany Comments». Definindo que esta associação irá permitir a nós costurar comentários a artigos quando o artigo é carregado.

Quando criada a tabela do seu banco de dados para uma relação hasMany, siga estas convenções:

hasMany: o *outro* modelo contem uma chave estrangeira.

Relação	Schema
Article hasMany Comment	Comment.article_id
Product hasMany Option	Option.product_id
Doctor hasMany Patient	Patient.doctor_id

Nós podemos definir uma associação hasMany em noos modelo de Articles seguindo:

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

Nós também podemos definir uma relação mais especifica usando os setters:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->hasMany('Comments')
            ->setForeignKey('article_id')
            ->setDependent(true);
    }
}
```

As vezes você pode querer configurar chaves compostas em sua associação:

```
// Within ArticlesTable::initialize() call
$this->hasMany('Reviews')
    ->setForeignKey([
        'article_id',
        'article_hash'
    ]);
```

Confiando no exemplo acima, nos passamos uma array contendo as chaves desejadas para `setForeignKey()`. Pos padrão o `bindingKey` seria automaticamente definido como `id` e `hash` respectivamente, mas vamos assimir que você precissa especificar diferentes campos como o padrão, vocẽ pdoe definir isto manualmente com `setBindingKey()`:

```
// Within ArticlesTable::initialize() call
$this->hasMany('Reviews')
    ->setForeignKey([
        'article_id',
        'article_hash'
    ])
    ->setBindingKey([
        'whatever_id',
        'whatever_hash'
    ]);
```

É importante perceber que os valores de `foreignKey` reference a tabela e `bindingKey` os valores reference a tabela **articles**.

As chaves possiveis para as associações hasMany são:

- **className**: o nome da classe do modelo associado a o modelo atual. Se você estiver definindo um "Usuário tem muitos comentários" relacionamento, a chave className deve ser igual a "Comentários".
- **foreignKey**: o nome da chave estrangeira encontrada no outro mesa. Isto é especialmente útil se você precisar definir múltiplos tem muitos relacionamentos. O valor padrão para esta chave é o sublinhado, nome singular do modelo real, sufixo com "_id".
- **bindingKey**: O nome da coluna na tabela atual, que será usado para combinar o `foreignKey`. Se não for especificado, a chave primária (para exemplo, a coluna de identificação da tabela `Artigos`) será usada.
- **conditions**: uma série de condições compatíveis com find () ou SQL strings como `` ["Comments.visible" => true] ``
- **sort**: uma série de cláusulas de pedido compatíveis com find () ou SQL strings como `` ["Comments.created" => "ASC"] ``
- **dependent**: Quando dependente é definido como `` true``, modelo recursivo a eliminação é possível. Neste exemplo, os registros de comentários serão excluído quando o registro do artigo associado foi excluído.

- **cascadeCallbacks**: Quando este e ** dependentes ** são `` true``, em cascata as exclusões carregarão e excluirão entidades para que as devoluções de chamada sejam corretamente desencadeada. Quando `` false``, `` deleteAll () `` é usado para remover dados associados e nenhuma devolução de chamada é desencadeada.
- **propertyName**: O nome da propriedade que deve ser preenchido com dados do Tabela associada aos resultados da tabela de origem. Por padrão, esta é a sublinhado e nome plural da associação para «comentários» no nosso exemplo.
- **strategy**: Define a estratégia de consulta a ser usada. Por padrão, selecione «selecionar». o outro valor válido é «subconsulta», que substitui a lista `` IN`` por uma subconsulta equivalente.
- **saveStrategy**: Ou «anexar» ou «substituir». Por padrão, «anexar». Quando "anexar" a corrente os registros são anexados a qualquer registro no banco de dados. Quando «substituir» associado Os registros que não estão no conjunto atual serão removidos. Se a chave estrangeira for anulável coluna ou se «dependente» é verdadeira, os registros serão órfãos.
- **finder**: O método do buscador a ser usado ao carregar registros associados.

Uma vez que essa associação foi definida, encontre operações na tabela Artigos pode conter os registros de comentários se eles existem:

```
// In a controller or table method.
$query = $articles->find('all')->contain(['Comments']);
foreach ($query as $article) {
    echo $article->comments[0]->text;
}
```

O anterior emitiria SQL que é semelhante ao:

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (1, 2, 3, 4, 5);
```

Quando a estratégia de subconsulta é usada, SQL semelhante ao seguinte será gerado:

```
SELECT * FROM articles;
SELECT * FROM comments WHERE article_id IN (SELECT id FROM articles);
```

Você pode querer armazenar em cache as contagens para suas associações hasMany. Isso é útil quando você costuma mostrar o número de registros associados, mas não deseja carregue todos os registros apenas para contabilizá-los. Por exemplo, o comentário conta com O artigo dado geralmente é armazenado em cache para tornar a geração de listas de artigos mais eficiente. Você pode usar o *CounterCacheBehavior* para armazenar contagens de registros associados.

Você deve certificar-se de que as tabelas do banco de dados não contenham colunas que correspondam nomes de propriedade da associação. Se, por exemplo, você tiver campos de contador que conflitam com propriedades de associação, você deve renomear a propriedade de associação ou o nome da coluna.

Associações BelongsToMany

Nota: Na versão 3.0 em diante `hasAndBelongsToMany` / `HABTM` foi renomeado para `belongsToMany` / `BTM`.

Um exemplo de uma associação BelongsToMany é «Article BelongsToMany Tags», onde as tags de um artigo são compartilhadas com outros artigos. BelongsToMany is muitas vezes referido como «tem e pertence a muitos», e é um clássico «muitos a muitos» Associação.

A principal diferença entre hasMany e BelongsToMany é que o link entre Os modelos de uma associação BelongsToMany não são exclusivos. Por exemplo, estamos juntando-se à nossa tabela de Artigos com uma tabela de Tags.

Usando “engraçado” como Tag para meu Artigo, não «use» a etiqueta. Eu também posso usá-lo no próximo artigo Eu escrevo.

São necessárias três tabelas de banco de dados para uma associação BelongsToMany. No exemplo acima, precisaríamos de tabelas para `` articles``, `` tags`` e `` articles_tags``. A tabela `` articles_tags`` contém os dados que ligam tags e artigos juntos. A tabela de junção é nomeada após as duas tabelas envolvido, separado com um sublinhado por convenção. Na sua forma mais simples, isso a tabela consiste em `` article_id`` e `` tag_id``.

belongsToMany requer uma tabela de junção separada que inclui ambos * modelo * nomes.

Relação	Join Table Fields
Article belongsToMany Tag	articles_tags.id, articles_tags.tag_id, articles_tags.article_id
Patient belongsToMany Doctor	doctors_patients.id, doctors_patients.doctor_id, doctors_patients.patient_id.

Podemos definir a associação belongsToMany em ambos os modelos da seguinte forma:

```
// In src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Tags');
    }
}

// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Articles');
    }
}
```

Também podemos definir uma relação mais específica usando a configuração:

```
// In src/Model/Table/TagsTable.php
class TagsTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Articles', [
            'joinTable' => 'articles_tags',
        ]);
    }
}
```

Chaves possiveis para uma associação pertence a muitos inclui:

- **className**: ome da classe do modelo associado a o modelo atual. Se você estiver definindo um “Artigo que pertence a outra etiqueta” relacionamento, a chave className deve igualar “Tags”.
- **joinTable**: O nome da tabela de junção usada neste associação (se a tabela atual não aderir à nomeação convenção para as mesas JoinToMany join). Por padrão, esta tabela O nome será usado para carregar a instância da tabela para a tabela de junção.

- **foreignKey**: O nome da chave estrangeira que faz referência ao modelo atual encontrado na tabela de junção, ou lista no caso de chaves externas compostas. Isto é especialmente útil se você precisar definir múltiplos pertence a muitos relacionamentos. O valor padrão para esta chave é o sublinhado, nome singular do modelo atual, sufixo com "_id".
- **bindingKey**: O nome da coluna na tabela atual, que será usado para combinar o `` foreignKey``. Predefinições para a chave primária.
- **targetForeignKey**: O nome da chave estrangeira que faz referência ao alvo modelo encontrado no modelo de junção, ou lista no caso de chaves externas compostas. O valor padrão para esta chave é o nome sublinhado, singular de o modelo alvo, sufixo com "_id".
- **conditions**: uma série de condições compatíveis com `find()`. Se você tem condições em uma tabela associada, você deve usar um modelo "through", e defina as participações necessárias para as associações nela.
- **sort**: uma série de cláusulas de ordem compatíveis com find ().
- **dependent**: Quando a chave dependente é definida como `` false``, e uma entidade é excluído, os dados da tabela de junção não serão excluídos.
- **through**: Permite que você forneça o apelido da instância da tabela que você quer usado na tabela de junção, ou a instância em si. Isso torna a personalização as chaves de tabela de junção possíveis e permite que você personalize o comportamento do tabela dinâmica.
- **cascadeCallbacks**: Quando isso é `` true``, os apagados em cascata serão carregados e Elimine entidades de modo que as devoluções de chamada sejam ativadas corretamente na tabela de junção registros. Quando `` false``, `` deleteAll () `` é usado para remover dados associados e nenhum retorno de chamada é desencadeado. Este padrão é «falso» para ajudar a reduzir a sobrecarga.
- **propertyName**: O nome da propriedade que deve ser preenchido com dados do Tabela associada aos resultados da tabela de origem. Por padrão, esta é a sublinhado e nome plural da associação, então «tags» no nosso exemplo.
- **strategy**: Define a estratégia de consulta a ser usada. Por padrão, selecione «selecionar». o outro valor válido é «subconsulta», que substitui a lista `` IN`` por uma subconsulta equivalente.
- **saveStrategy**:Ou «anexar» ou «substituir». Por padrão, "substituir". Indica o modo a ser usado para salvar entidades associadas. O primeiro apenas crie novas ligações entre ambos os lados da relação e o último faça uma limpeza e substitua para criar os links entre as entidades aprovadas quando poupança.
- **finder**: O método do buscador a ser usado ao carregar registros associados.

Uma vez definida esta associação, encontrar operações na tabela Artigos podem conter os registros de tags se eles existirem:

```
// Em um método de controlador ou tabela.
$query = $articles->find('all')->contain(['Tags']);
foreach ($query as $article) {
    echo $article->tags[0]->text;
}
```

O anterior emitiria SQL que é semelhante ao:

```
SELECT * FROM articles;
SELECT * FROM tags
INNER JOIN articles_tags ON (
  tags.id = article_tags.tag_id
  AND article_id IN (1, 2, 3, 4, 5)
);
```

Quando a estratégia de subconsulta é usada, SQL semelhante ao seguinte será gerado:

```
SELECT * FROM articles;
SELECT * FROM tags
INNER JOIN articles_tags ON (
  tags.id = article_tags.tag_id
  AND article_id IN (SELECT id FROM articles)
);
```

Usando a opção «através»

Se você planeja adicionar informações adicionais à tabela de junção / pivô, ou se precisar Para usar juntas colunas fora das convenções, você precisará definir a `` through``. A opção `` through`` oferece controle total sobre como A associação belongsToMany será criada..

Às vezes, é desejável armazenar dados adicionais com muitos Associação. Considere o seguinte:

```
Student BelongsToMany Course
Course BelongsToMany Student
```

Um aluno pode fazer muitos cursos e um curso pode ser realizado por muitos estudantes. este é uma associação simples de muitos a muitos. A tabela a seguir seria suficiente:

```
id | student_id | course_id
```

Agora, e se quisermos armazenar o número de dias que foram atendidos pelo Estudante no curso e a nota final? A tabela que queremos seria:

```
id | student_id | course_id | days_attended | grade
```

A maneira de implementar nosso requisito é usar um **join model**, de outra forma conhecido como um **hasMany through** Associação. Ou seja, a associação é um modelo em si. Então, podemos criar um novo modelo de CursosMemberships. Dê uma olhada no seguintes modelos:

```
class StudentsTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Courses', [
            'through' => 'CoursesMemberships',
        ]);
    }
}

class CoursesTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsToMany('Students', [
            'through' => 'CoursesMemberships',
        ]);
    }
}
```

(continues on next page)

(continuação da página anterior)

```
class CoursesMembershipsTable extends Table
{
    public function initialize(array $config)
    {
        $this->belongsTo('Students');
        $this->belongsTo('Courses');
    }
}
```

A tabela de junção CoursesMemberships identifica de forma exclusiva um determinado Student's Participação em um Curso, além de meta-informação extra.

Condições de associação padrão

A opção `` finder`` permite que você use um :ref:` finder customizado <custom-find-methods>` para carregar registros associados. Isto permite a você encapsular as suas queries melhor e manter o seu codigo mais enxuto. Existem algumas limitações quando usando finders para carregar dados em associações que são carregadas usando junções (belongsTo/hasOne).Somente os seguintes aspectos da consulta serão aplicados a a consulta do raiz:

- condições WHERE.
- joins adicionais.
- Associações contidas.

Outros aspectos da consulta, tais como colunas selecionadas, ordem, grupo, tendo e outras sub-declarações, não serão aplicadas à consulta raiz. Associações que * não * são carregados através de associações (hasMany / belongsToMany), não têm o acima das restrições e também pode usar formatadores de resultados ou mapa / reduzir funções.

Carregando Associações

Uma vez que você definiu as suas associações, você pode *associações de carga ansiosas* ao obter resultados.

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 connectem 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 especifica 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()`:

```
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

class Cake\ORM\Behavior**CounterCacheBehavior**

Muitas vezes, os aplicativos da web precisam exibir contagens de objetos relacionados. Por exemplo, ao mostrar uma lista de artigos, você pode exibir quantos comentários ela possui. Ou, ao mostrar a um usuário, você pode mostrar quantos amigos/seguidores ele tem. O comportamento do CounterCache é destinado a essas situações. O CounterCache atualizará um campo nos modelos associados atribuídos nas opções quando for chamado. Os campos devem existir no banco de dados e ser do tipo INT.

Uso Básico

Você ativa o comportamento do CounterCache como qualquer outro comportamento, mas ele não fará nada até você configurar algumas relações e as contagens de campos que devem ser armazenadas em cada uma delas. Usando nosso exemplo abaixo, poderíamos armazenar em cache a contagem de comentários para cada artigo com o seguinte:

```
class CommentsTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('CounterCache', [
            'Articles' => ['comment_count']
        ]);
    }
}
```

A configuração do CounterCache deve ser um mapa dos nomes das relações e a configuração específica para essa relação.

O valor do contador será atualizado sempre que uma entidade for salva ou excluída. O contador **não** será atualizado quando você usar `updateAll()` ou `deleteAll()`, ou executar o SQL que você escreveu.

Uso Avançado

Se você precisar manter um contador em cache por menos que todos os registros relacionados, poderá fornecer condições adicionais ou métodos localizadores para gerar um valor de contador:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'finder' => 'published'
        ]
    ]
]);
```

Se você não tiver um método localizador personalizado, poderá fornecer uma variedade de condições para localizar registros:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'conditions' => ['Comments.spam' => false]
        ]
    ]
]);
```

Se você deseja que o CounterCache atualize vários campos, por exemplo, mostrando uma contagem condicional e uma contagem básica, você pode adicionar esses campos na matriz:

```
$this->addBehavior('CounterCache', [
    'Articles' => ['comment_count',
        'published_comment_count' => [
            'finder' => 'published'
```

(continues on next page)

(continuação da página anterior)

```
        ]
    ]
]);
```

Se você deseja calcular o valor do campo CounterCache por conta própria, pode definir a opção `ignoreDirty` como `true`.

Isso impedirá que o campo seja recalculado se você o tiver definido antes:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'comment_count' => [
            'ignoreDirty' => true
        ]
    ]
]);
```

Por fim, se um localizador personalizado e as condições não forem adequadas, você poderá fornecer uma função de retorno de chamada. Sua função deve retornar o valor da contagem a ser armazenada:

```
$this->addBehavior('CounterCache', [
    'Articles' => [
        'rating_avg' => function ($event, $entity, $table, $original) {
            return 4.5;
        }
    ]
]);
```

Sua função pode retornar `false` para pular a atualização da coluna do contador ou um objeto `Query` que produziu o valor da contagem. Se você retornar um objeto `Query`, sua consulta será usada como uma subconsulta na instrução de atualização. O parâmetro `$table` refere-se ao objeto de tabela que mantém o comportamento (não a relação de destino) por conveniência. O retorno de chamada é chamado pelo menos uma vez com `$original` definido como `false`. Se a atualização da entidade alterar a associação, o retorno de chamada será invocado uma *segunda* vez com `true`, o valor de retorno atualizará o contador do item associado *anteriormente*.

Nota: O comportamento do CounterCache funciona apenas para associações `belongsTo`. Por exemplo, para «Comentários pertence a artigos», é necessário adicionar o comportamento do CounterCache ao `CommentsTable` para gerar `comment_count` para a tabela Articles.

É possível, no entanto, fazer isso funcionar para associações `belongsToMany`. Você precisa habilitar o comportamento do CounterCache em uma tabela personalizada `through` configurada nas opções de associação e definir a opção de configuração `cascadeCallbacks` como true. Veja como configurar uma tabela de junção personalizada *Usando a opção «através»*.

Alterado na versão 3.6.0: Retornando `false` para pular as atualizações foi adicionado.

Timestamp

class Cake\ORM\Behavior**TimestampBehavior**

O behavior timestamp permite que seus objetos atualizem um ou mais timestamps a cada evento. É utilizado principalmente para preencher os campos `created` e `modified`. Entretanto, com algumas configurações adicionais, você pode atualizar qualquer coluna timestamp/datetime de qualquer tabela.

Uso Básico

Você ativa o behavior timestamp como qualquer outro behavior:

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

A configuração padrão fará o seguinte:

- Quando uma nova entidade é salva, os campos `created` e `modified` são preenchidos com a hora atual.
- Quando uma entidade é atualizada, o campo `modified` é preenchido com a hora atual.

Usando e Configurando o Behavior

Se você precisar modificar campos com nomes diferentes ou quiser atualizar campos de timestamp em eventos personalizados, você pode usar uma configuração adicional:

```
class OrdersTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp', [
            'events' => [
                'Model.beforeSave' => [
                    'created_at' => 'new',
                    'updated_at' => 'always',
                ],
                'Orders.completed' => [
                    'completed_at' => 'always'
                ]
            ]
        ]);
    }
}
```

Como você pode ver acima, além do evento padrão `Model.beforeSave`, nós também estamos atualizando a coluna `completed_at` quando os pedidos são concluídos.

Atualizando Timestamps em Entidades

Às vezes você vai querer atualizar apenas os timestamps em uma entidade sem alterar quaisquer outras propriedades. Isso é chamado de "touching" uma informação. No CakePHP você pode usar o método `touch()` para fazer exatamente isso:

```
// Touch baseado no evento Model.beforeSave.
$articles->touch($article);

// Touch baseado em um evento específico.
$orders->touch($order, 'Orders.completed');
```

Após você salvar a entidade, o campo é atualizado.

Registro "touching" pode ser útil quando você deseja avisar que um recurso pai mudou quando um recurso filho é criado/atualizado. Por exemplo: atualizar um artigo quando um novo comentário é adicionado.

Salvando Atualizações Sem Alterar o Timestamp

Para desativar a atualização automática do campo `updated`, quando salvamos uma entidade, você pode marcar o atributo como "dirty":

```
// Marca a coluna modified como dirty
$order->setDirty('modified', true);

// Anterior a versão 3.4.0
$order->dirty('modified', true);
```

Translate

class Cake\ORM\Behavior**TranslateBehavior**

O comportamento Translate permite criar e recuperar cópias traduzidas de suas entidades em vários idiomas. Isso é feito usando uma tabela `i18n` separada, na qual armazena a tradução para cada um dos campos de qualquer objeto de Tabela ao qual está vinculado.

> **Aviso:** O TranslateBehavior não suporta chaves primárias compostas neste momento.

Um Tour Rápido

Depois de criar a tabela `i18n` no seu banco de dados, anexe o comportamento a qualquer objeto da Tabela que você deseja tornar traduzível:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', ['fields' => ['title']]);
    }
}
```

Agora, selecione um idioma a ser usado para recuperar entidades alterando o idioma do aplicativo, o que afetará todas as traduções:

```
// Em um controlador. Alterar a localidade, por exemplo para espanhol
I18n::setLocale('es');
$this->loadModel('Articles');
```

Em seguida, obtenha uma entidade existente:

```
$article = $this->Articles->get(12);
echo $article->title; // Eco em 'Um título', ainda não traduzido
```

Em seguida, traduza sua entidade:

```
$article->title = 'Un Artículo';
$this->Articles->save($article);
```

Agora você pode tentar obter sua entidade novamente:

```
$article = $this->Articles->get(12);
echo $article->title; // Echo em 'Un Artículo', yay pedaço de bolo!
```

O trabalho com várias traduções pode ser feito usando uma trait especial na sua classe Entity:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Agora você pode encontrar todas as traduções para uma única entidade:

```
$article = $this->Articles->find('translations')->first();
echo $article->translation('es')->title; // 'Un Artículo'

echo $article->translation('en')->title; // 'An Article';
```

É igualmente fácil salvar várias traduções ao mesmo tempo:

```
$article->translation('es')->title = 'Otro Título';
$article->translation('fr')->title = 'Un autre Titre';
$this->Articles->save($article);
```

Se você quiser se aprofundar em como ele funciona ou como ajustar o comportamento às suas necessidades, continue lendo o restante deste capítulo.

Inicializando a Tabela do Banco de Dados i18n

Para usar o comportamento, você precisa criar uma tabela `i18n` com o esquema correto. Atualmente, a única maneira de carregar a tabela `i18n` é executando manualmente o seguinte script SQL no seu banco de dados:

```
CREATE TABLE i18n (
    id int NOT NULL auto_increment,
    locale varchar(6) NOT NULL,
    model varchar(255) NOT NULL,
    foreign_key int(10) NOT NULL,
    field varchar(255) NOT NULL,
    content text,
    PRIMARY KEY     (id),
    UNIQUE INDEX I18N_LOCALE_FIELD(locale, model, foreign_key, field),
    INDEX I18N_FIELD(model, foreign_key, field)
);
```

O esquema também está disponível como arquivo sql em **/config/schema/i18n.sql**.

Uma observação sobre as abreviações de idioma: O comportamento de tradução não impõe restrições ao identificador de idioma; os valores possíveis são restritos apenas pelo tipo/tamanho da coluna `locale`. `locale` é definido como `varchar(6)` caso você queira usar abreviações como `es-419` (espanhol para América Latina, abreviação de idioma com código de área UN M.49[90]).

Dica: É aconselhável usar as mesmas abreviações de idioma necessárias para *Internacionalização e localização*. Assim, é consistente e a alternância do idioma funciona de forma idêntica para ambos, o `Translate Behaviour` e `Internationalization and Localization`.

Portanto, é recomendável usar o código ISO de duas letras do idioma como `en`, `fr`, `de` ou o nome completo da localidade, como `fr_FR`, `es_AR` , `da_DK`, que contém o idioma e o país em que é falado.

Anexando o Comportamento da Conversão às suas Tabelas

Anexar o comportamento pode ser feito no método `initialize()` na sua classe Table:

```
class ArticlesTable extends Table
{

    public function initialize(array $config)
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}
```

A primeira coisa a observar é que você deve passar a chave `fields` na matriz de configuração. Essa lista de campos é necessária para informar ao comportamento quais colunas serão capazes de armazenar traduções.

[90] https://en.wikipedia.org/wiki/UN_M.49

Usando uma Tabela de Traduções Separada

Se você deseja usar uma tabela diferente de `i18n` para converter um repositório específico, pode especificar o nome da classe da tabela para sua tabela personalizada na configuração do comportamento. Isso é comum quando você tem várias tabelas para traduzir e deseja uma separação mais limpa dos dados armazenados para cada tabela diferente:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'translationTable' => 'ArticlesI18n'
        ]);
    }
}
```

Você precisa ter certeza de que qualquer tabela personalizada usada tenha as colunas `field`, `foreign_key`, `locale` e `model`.

Lendo Conteúdo Traduzido

Como mostrado acima, você pode usar o método `setLocale()` para escolher a tradução ativa para entidades que são carregadas:

```
// Carregue as funções principais do I18n no início do seu Controller:
use Cake\I18n\I18n;

// Depois, você pode alterar o idioma em sua ação:
I18n::setLocale('es');
$this->loadModel('Articles');

// Todas as entidades nos resultados conterão tradução para o espanhol
$results = $this->Articles->find()->all();
```

Este método funciona com qualquer localizador em suas tabelas. Por exemplo, você pode usar o TranslateBehavior com `find('list')`:

```
I18n::setLocale('es');
$data = $this->Articles->find('list')->toArray();

// Os dados conterão
[1 => 'Mi primer artículo', 2 => 'El segundo artículo', 15 => 'Otro articulo' ...]
```

Recuperar Todas as Traduções para uma Entidade

Ao criar interfaces para atualizar o conteúdo traduzido, geralmente é útil mostrar uma ou mais traduções ao mesmo tempo. Você pode usar o localizador de `traduções` para isso:

```
// Encontre o primeiro artigo com todas as traduções correspondentes
$article = $this->Articles->find('translations')->first();
```

No exemplo acima, você receberá uma lista de entidades que possuem um conjunto de propriedades `_translations`. Esta propriedade conterá uma lista de entidades de dados de conversão. Por exemplo, as seguintes propriedades estariam acessíveis:

```
// Saídas 'en'
echo $article->_translations['en']->locale;

// Saídas 'title'
echo $article->_translations['en']->field;

// Saídas 'My awesome post!'
echo $article->_translations['en']->body;
```

Uma maneira mais elegante de lidar com esses dados é adicionando uma trait à classe de entidade usada para sua tabela:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Essa trait contém um único método chamado `translation`, que permite acessar ou criar novas entidades de tradução em tempo real:

```
// Saída de 'title'
echo $article->translation('en')->title;

// Adiciona uma nova entidade de dados de tradução ao artigo
$article->translation('de')->title = 'Wunderbar';
```

Limitando as Traduções a serem Recuperadas

Você pode limitar os idiomas buscados no banco de dados para um conjunto específico de registros:

```
$results = $this->Articles->find('translations', [
    'locales' => ['en', 'es']
]);
$article = $results->first();
$spanishTranslation = $article->translation('es');
$englishTranslation = $article->translation('en');
```

Impedindo a Recuperação de Traduções Vazias

Os registros de tradução podem conter qualquer sequência, se um registro tiver sido traduzido e armazenado como uma sequência vazia (“”), o comportamento da conversão será usado e o substituirá pelo valor do campo original.

Se isso for indesejável, você pode ignorar as traduções vazias usando a chave de configuração `allowEmptyTranslations`:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', [
            'fields' => ['title', 'body'],
            'allowEmptyTranslations' => false
        ]);
    }
}
```

O exemplo acima carregaria apenas os dados traduzidos que tenham conteúdo.

Recuperando Todas as Traduções para Associações

Também é possível encontrar traduções para qualquer associação em uma única operação de localização:

```
$article = $this->Articles->find('translations')->contain([
    'Categories' => function ($query) {
        return $query->find('translations');
    }
])->first();

// Saídas 'Programación'
echo $article->categories[0]->translation('es')->name;
```

Isso pressupõe que `Categories` tem o TranslateBehavior associado. Simplesmente use a função de construtor de consultas `contain` para usar o find personalizado `translator` na associação.

Recuperando um Idioma Sem Usar I18n::SetLocale

chamando `I18n::setLocale('es');` altera a localidade padrão para todas as descobertas traduzidas, pode haver momentos em que você deseja recuperar o conteúdo traduzido sem modificar o estado do aplicativo. Para esses cenários, use o método `setLocale()` do comportamento:

```
I18n::setLocale('en'); // redefinir para ilustração

$this->loadModel('Articles');

// localidade específica. Use locale () anterior à 3.6
$this->Articles->setLocale('es');

$article = $this->Articles->get(12);
echo $article->title; // Echoes 'Un Artículo', yay piece of cake!
```

Observe que isso altera apenas a localidade da tabela Articles, isso não afetaria o idioma dos dados associados. Para afetar os dados associados, é necessário chamar o método em cada tabela, por exemplo:

```
I18n::setLocale('en'); // Reseta para ilustração

$this->loadModel('Articles');
// Use locale() anterior à 3.6
$this->Articles->setLocale('es');
$this->Articles->Categories->setLocale('es');

$data = $this->Articles->find('all', ['contain' => ['Categories']]);
```

Este exemplo também pressupõe que `Categories` tem o TranslateBehavior associado.

Consultando Campos Traduzidos

TranslateBehavior não substitui as condições de localização por padrão. Você precisa usar o método `translationField()` para compor as condições de localização nos campos traduzidos:

```
// Use locale() anterior à 3.6
$this->Articles->setLocale('es');
$data = $this->Articles->find()->where([
    $this->Articles->translationField('title') => 'Otro Título'
]);
```

Salvando em Outro Idioma

A filosofia por trás do TranslateBehavior é que você tem uma entidade que representa o idioma padrão e várias traduções que podem substituir determinados campos nessa entidade. Tendo isso em mente, você pode salvar intuitivamente traduções para qualquer entidade. Por exemplo, dada a seguinte configuração:

```
// Em src/Model/Table/ArticlesTable.php
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', ['fields' => ['title', 'body']]);
    }
}

// Em src/Model/Entity/Article.php
class Article extends Entity
{
    use TranslateTrait;
}

// Em um Controller
$this->loadModel('Articles');
$article = new Article([
    'title' => 'My First Article',
    'body' => 'This is the content',
```

(continues on next page)

(continuação da página anterior)

```
    'footnote' => 'Some afterwords'
]);

$this->Articles->save($article);
```

Portanto, depois de salvar seu primeiro artigo, você pode salvar uma tradução para ele, existem algumas maneiras de fazê-lo. O primeiro é definir o idioma diretamente na entidade:

```
$article->_locale = 'es';
$article->title = 'Mi primer Artículo';

$this->Articles->save($article);
```

Depois que a entidade tiver sido salva, o campo traduzido também será mantido. Um valor a ser observado é que os valores do idioma padrão que não foram substituídos serão preservados:

```
// Saídas 'Este é o conteúdo'
echo $article->body;

// Saídas 'Mi primer Artículo'
echo $article->title;
```

Depois de substituir o valor, a conversão para esse campo será salva e poderá ser recuperada como de costume:

```
$article->body = 'El contendio';
$this->Articles->save($article);
```

A segunda maneira de usar para salvar entidades em outro idioma é definir o idioma padrão diretamente para a tabela:

```
$article->title = 'Mi Primer Artículo';

// Use locale() anterior à 3.6
$this->Articles->setLocale('es');
$this->Articles->save($article);
```

Definir o idioma diretamente na tabela é útil quando você precisa recuperar e salvar entidades para o mesmo idioma ou quando você precisa salvar várias entidades ao mesmo tempo.

Salvando Várias Traduções

É um requisito comum poder adicionar ou editar várias traduções em qualquer registro do banco de dados ao mesmo tempo. Isso pode ser feito usando o `TranslateTrait`:

```
use Cake\ORM\Behavior\Translate\TranslateTrait;
use Cake\ORM\Entity;

class Article extends Entity
{
    use TranslateTrait;
}
```

Agora, você pode preencher as traduções antes de salvá-las:

```
$translations = [
    'fr' => ['title' => "Un article"],
    'es' => ['title' => 'Un artículo']
];

foreach ($translations as $lang => $data) {
    $article->translation($lang)->set($data, ['guard' => false]);
}

$this->Articles->save($article);
```

A partir do 3.3.0, o trabalho com várias traduções foi simplificado. Você pode criar controles de formulário para seus campos traduzidos:

```
// Em um template.
<?= $this->Form->create($article); ?>
<fieldset>
    <legend>French</legend>
    <?= $this->Form->control('_translations.fr.title'); ?>
    <?= $this->Form->control('_translations.fr.body'); ?>
</fieldset>
<fieldset>
    <legend>Spanish</legend>
    <?= $this->Form->control('_translations.es.title'); ?>
    <?= $this->Form->control('_translations.es.body'); ?>
</fieldset>
```

No seu controlador, você pode organizar os dados normalmente:

```
$article = $this->Articles->newEntity($this->request->getData());
$this->Articles->save($article);
```

Isso resultará no seu artigo, que todas as traduções em francês e espanhol serão mantidas. Você precisará se lembrar de adicionar `_translations` aos campos `$ _accessible` da sua entidade também.

Validando Entidades Traduzidas

Ao anexar `TranslateBehavior` a um modelo, você pode definir o validador que deve ser usado quando os registros de conversão são criados/modificados pelo comportamento durante `newEntity()` ou `patchEntity()`:

```
class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Translate', [
            'fields' => ['title'],
            'validator' => 'translated'
        ]);
    }
}
```

O texto acima usará o validador criado por `validationTranslated` para entidades traduzidas validadas.

Árvore

class Cake\ORM\Behavior**TreeBehavior**

É bastante comum querer armazenar dados hierárquicos em uma tabela no banco de dados. Exemplos de tais dados podem ser categorias com subcategorias, dados relacionados a um sistema de menu multinível ou uma representação literal de hierarquia, como departamentos em uma empresa.

Bancos de dados relacionais não são adequados para armazenar e recuperar esses tipos de dados, mas existem algumas técnicas conhecidas que podem torná-los eficazes para trabalhar com informações de vários níveis.

O TreeBehavior ajuda a manter uma estrutura de dados hierárquica no banco de dados, que pode ser consultado sem muita sobrecarga e ajuda a reconstruir os dados da árvore.

Requisitos

Esse behavior requer as seguintes colunas na tabela do seu banco de dados:

- `parent_id` (nullable) A coluna que contém o ID da linha pai
- `lft` (integer, signed) Usado para manter a estrutura da árvore
- `rght` (integer, signed) Usado para manter a estrutura da árvore

Você pode configurar o nome desses campos caso precise personalizá-los. Mais informações sobre o significado dos campos e como elas são usadas podem ser encontradas neste artigo que descreve a lógica do MPTT[91]

Aviso: O TreeBehavior não suporta chaves primárias compostas.

Início rápido

Você ativa o comportamento da árvore, adicionando-o a tabela que deseja armazenar dados hierárquicos:

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

Uma vez adicionado, você pode deixar o CakePHP construir a estrutura interna se a tabela já estiver pronta:

```
// Prior to 3.6 use TableRegistry::get('Articles')
$categories = TableRegistry::getTableLocator()->get('Categories');
$categories->recover();
```

Você pode verificar se funciona, obtendo qualquer linha da tabela e pedindo a contagem de descendentes que ela tem:

```
$node = $categories->get(1);
echo $categories->childCount($node);
```

[91] https://www.sitepoint.com/hierarchical-data-database-2/

Obter uma lista simples dos descendentes de um nó é igualmente fácil:

```
$descendants = $categories->find('children', ['for' => 1]);

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

Se você precisar informar algumas condições:

```
$descendants = $categories
    ->find('children', ['for' => 1])
    ->where(['name LIKE' => '%Foo%']);

foreach ($descendants as $category) {
    echo $category->name . "\n";
}
```

Se você precisar de uma lista encadeada, onde os filhos de cada nó são aninhados em uma hierarquia, você pode utilizar o localizador «threaded»:

```
$children = $categories
    ->find('children', ['for' => 1])
    ->find('threaded')
    ->toArray();

foreach ($children as $child) {
    echo "{$child->name} has " . count($child->children) . " direct children";
}
```

Percorrer resultados encadeados geralmente requer funções recursivas, mas se você precisa apenas um conjunto de resultados contendo um único campo de cada nível, para que você possa exibir um «select list» em seu formulário HTML, por exemplo, é melhor usar o localizador “treeList”

```
$list = $categories->find('treeList');

// In a CakePHP template file:
echo $this->Form->control('categories', ['options' => $list]);

// Or you can output it in plain text, for example in a CLI script
foreach ($list as $categoryName) {
    echo $categoryName . "\n";
}
```

A saída será similar a essa:

```
My Categories
_Fun
__Sport
___Surfing
___Skating
_Trips
__National
__International
```

O localizador `treeList` tem várias opções:

- `keyPath`: Um caminho separado por ponto para buscar o campo a ser usado pela chave do array, ou um closure para retornar a chave da linha fornecida.
- `valuePath`: Um caminho separado por ponto para buscar o campo a ser usado para o valor do array, ou um closure para retornar o valor da linha fornecida.
- `spacer`: Uma string a ser usada como prefixo para demonstrar a profundidade na árvore para cada item.

Um exemplo de todas as opções em uso é:

```
$query = $categories->find('treeList', [
    'keyPath' => 'url',
    'valuePath' => 'id',
    'spacer' => ' '
]);
```

Uma tarefa muito comum é encontrar o caminho de um determinado nó para a raiz da árvore. Isso é útil, por exemplo, para adicionar a lista de breadcrumbs em uma estrutura de menu:

```
$nodeId = 5;
$crumbs = $categories->find('path', ['for' => $nodeId]);

foreach ($crumbs as $crumb) {
    echo $crumb->name . ' > ';
}
```

Árvores construídas com o TreeBehavior não podem ser ordenadas por nenhuma outra coluna que não seja a coluna `` lft``, isso ocorre porque a representação interna da árvore depende dessa classificação. Felizmente, você pode reordenar os nós dentro do mesmo nível sem ter que mudar de pai:

```
$node = $categories->get(5);

// Move o nó uma posição para cima
$categories->moveUp($node);

// Move o nó para o topo da lista dentro do mesmo nível.
$categories->moveUp($node, true);

// Move o nó para o fundo.
$categories->moveDown($node, true);
```

Configuração

Se os nomes padrões utilizados pelo Behavior não correspondem aos nomes utilizados na sua tabela, você pode adicionar apelidos a eles:

```
public function initialize(array $config)
{
    $this->addBehavior('Tree', [
        'parent' => 'ancestor_id', // Use isso em vez de parent_id
        'left' => 'tree_left', // Use isso em vez de lft
        'right' => 'tree_right' // Use isso em vez de rght
```

(continues on next page)

(continuação da página anterior)

```
    ]);
}
```

Nível do nó (Profundidade)

Conhecer a profundida dos nós da árvore pode ser úti quando você precisa recuperar todos os nós até um certo nível para, por exemplo, gerar menus. Você pode usar a opção `level` para especificar o campo que irá guardar o nível de cada nó:

```
$this->addBehavior('Tree', [
    'level' => 'level', // O padrão é null, ou seja, não salva o nível
]);
```

Se você não quiser armazenar o nível em cache, você pode usar o método `TreeBehavior::getLevel()` para saber o nível do nó.

Escopo e Multi Árvores

Às vezes você precisa guardar mais de uma árvore dentro da mesma tabela, você pode conseguir isso usando a configuração "scope". Por exemplo, em uma tabela de localizações, você pode querer criar uma árvore por país:

```
class LocationsTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Tree', [
            'scope' => ['country_name' => 'Brazil']
        ]);
    }
}
```

No exemplo anterior, todas as operações de árvore terão o escopo apenas para as linhas que tem a coluna `country_name` definida como "Brazil". Você pode mudar o escopo utilizando a função "config":

```
$this->behaviors()->Tree->config('scope', ['country_name' => 'France']);
```

Opcionalmente, você pode ter um controle mais refinado do escopo passando um closure como o escopo:

```
$this->behaviors()->Tree->config('scope', function ($query) {
    $country = $this->getConfigureContry(); // uma função inventada
    return $query->where(['country_name' => $country]);
});
```

Recuperando com campo de classificação personalizada

Por padrão, recover() classifica os itens utilizando a chave primária. Isso funciona muito bem se a chave primária é uma coluna numérica (incremento automático), mas pode levar a resultados estranhos se você use UUIDs.

Se você precisar de classificação personalizada, você pode definir uma cláusula de ordem personalizada na sua configuração:

```
$this->addBehavior('Tree', [
    'recoverOrder' => ['country_name' => 'DESC'],
]);
```

Salvando Dados Hierárquicos

Ao usar o Tree Behavior, você geralmente não precisa se preocupar com a representação interna da estrutura hierárquica. As posições em que os nós são colocados na árvore são deduzidos a partir da coluna "parent_id" em cada um das suas entidades:

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = 5;
$categoriesTable->save($aCategory);
```

Fornecer ids para um nó pai não existente ao salvar ou ao tentar criar um loop na árvore (fazendo um nó filho de si mesmo) lançará uma exceção.

Você pode transformar um nó em uma raiz na árvore definindo a coluna "parent_id" como null:

```
$aCategory = $categoriesTable->get(10);
$aCategory->parent_id = null;
$categoriesTable->save($aCategory);
```

Os filhos do novo nó raiz serão preservados.

Apagando nós

Excluir um nó e toda a sua sub-árvore (qualquer nó filho que esteja em profundidade na árvore) é trivial:

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->delete($aCategory);
```

O TreeBehavior cuidará de todas as operações internas de exclusão para você. Também é possível excluir apenas um nó e reatribuir todos os filhos ao nó pai imediatamente superior na árvore

```
$aCategory = $categoriesTable->get(10);
$categoriesTable->removeFromTree($aCategory);
$categoriesTable->delete($aCategory);
```

Todos os nós filhos serão mantidos e um novo pai será atribuído a eles.

A exclusão de um nó é baseada nos valores lft e rght da entidade. Isto é importante quando estamos fazendo um loop através dos filhos de um nó para exclusões condicionais:

```
$descendants = $teams->find('children', ['for' => 1]);

foreach ($descendants as $descendant) {
    $team = $teams->get($descendant->id);
    if ($team->expired) {
        $teams->delete($team); // a exclusão reordena o lft e o rght no banco de dados
    }
}
```

O TreeBehavior reordena os valores lft e rght dos registros na tabela quando um nó foi deletado. Como tal, os valores lft e rght das entidades dentro de `` $ descendants`` (salvo antes da operação de exclusão) será impreciso. Entidades terão que ser carregadas e modificadas em tempo real para evitar inconsistências na tabela.

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 convensão 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
{
}
```

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
```

(continues on next page)

(continuação da página anterior)

```
{

    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 salvarmos entidades nós podemos realizar slug automaticamento 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 mesmo 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',
```

(continues on next page)

(continuação da página anterior)

```
    ]
]);
```

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 convensões 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 Metódos 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 deixara 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.

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:

```php
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 informaçes 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:

```php
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', [
```

(continues on next page)

(continuação da página anterior)

```
  '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 equivalete, 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 restriçes não podem ser adicionados para colunas que não existem, já que isso resultaria em um estado ínvalido. Os índices são difentes 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', [
```

(continues on next page)

(continuação da página anterior)

```
  '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 contêm 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
```

(continues on next page)

(continuação da página anterior)

```
$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 indices. Algumas plataformas podem requerer várias declaraçes 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.
// Prior to 3.4 use $db->schemaCollection()
$collection = $db->getSchemaCollection();

// 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>>
```

CAPÍTULO 13

Bake Console

Esta página foi movida[92].

[92] https://book.cakephp.org/bake/1.x/pt/

CAPÍTULO 14

Caching

class Cake\Cache**Cache**

O cache pode ser usado para acelerar a leitura de recursos caros ou lentos, mantendo uma segunda cópia dos dados necessários em um sistema de armazenamento mais rápido ou mais próximo. Por exemplo, você pode armazenar os resultados de consultas caras ou acesso remoto ao serviço da Web que não muda com freqüência em um cache. Uma vez no cache, é muito mais barato do que acessar o recurso remoto.

O armazenamento em cache no CakePHP é facilitado pela classe `Cache`. Esta classe fornece uma interface estática e uma API uniforme para interagir com várias implementações de armazenamento em cache. O CakePHP fornece vários mecanismos de cache e fornece uma interface simples se você precisar criar seu próprio back-end. Os mecanismos de armazenamento em cache integrados são:

- `File` O cache de arquivo é um cache simples que usa arquivos locais. É o mecanismo de cache mais lento e não fornece tantos recursos para operações atômicas. No entanto, como o armazenamento em disco geralmente é bastante barato, o armazenamento de objetos grandes ou elementos que raramente são gravados funciona bem em arquivos.
- `Memcached` Usa a extensão Memcached[93].
- `Redis` Usa a extensão phpredis[94]. O Redis fornece um sistema de cache rápido e persistente semelhante ao Memcached, também fornece operações atômicas.
- `Apcu` O cache do APCu usa a extensão PHP APCu[95]. Essa extensão usa memória compartilhada no servidor da web para armazenar objetos. Isso o torna muito rápido e capaz de fornecer recursos atômicos de leitura/gravação.
- `Wincache` O Wincache usa a extensão Wincache[96]. O Wincache é semelhante ao APC em recursos e desempenho, mas otimizado para Windows e IIS.
- `Array` Armazena todos os dados em uma matriz. Esse mecanismo não fornece armazenamento persistente e deve ser usado em conjuntos de testes de aplicativos.

[93] https://php.net/memcached
[94] https://github.com/phpredis/phpredis
[95] https://php.net/apcu
[96] https://php.net/wincache

- `Null` O mecanismo nulo não armazena nada e falha em todas as operações de leitura.

Independentemente do CacheEngine que você escolher, seu aplicativo interage com `Cake\Cache\Cache`.

Configurando Mecanismos de Cache

static Cake\Cache\Cache::**setConfig**(*$key*, *$config = null*)

Seu aplicativo pode configurar qualquer número de "engines" durante o processo de inicialização. As configurações do mecanismo de cache são definidas em **config/app.php**.

Para um desempenho ideal, o CakePHP requer que dois mecanismos de cache sejam definidos.

- `_cake_core_` é usado para armazenar mapas de arquivos e resultados analisados de arquivos *Internacionalização e Localização*.
- `_cake_model_`, é usado para armazenar descrições de esquema para seus modelos de aplicativos.

O uso de várias configurações de mecanismo também permite alterar gradualmente o armazenamento, conforme necessário. Por exemplo, em seu **config/app.php**, você pode colocar o seguinte:

```
// ...
'Cache' => [
    'short' => [
        'className' => 'File',
        'duration' => '+1 hours',
        'path' => CACHE,
        'prefix' => 'cake_short_'
    ],
    // Usando um namespace para nome completo.
    'long' => [
        'className' => 'Cake\Cache\Engine\FileEngine',
        'duration' => '+1 week',
        'probability' => 100,
        'path' => CACHE . 'long' . DS,
    ]
]
// ...
```

As opções de configuração também podem ser fornecidas como uma string *DSN*. Isso é útil ao trabalhar com variáveis de ambiente ou provedores de *PaaS*:

```
Cache::setConfig('short', [
    'url' => 'memcached://user:password@cache-host/?timeout=3600&prefix=myapp_',
]);
```

Ao usar uma sequência DSN, você pode definir parâmetros/opções adicionais como argumentos da sequência de consultas.

Você também pode configurar os mecanismos de cache em tempo de execução:

```
// Usando um nome abreviado
Cache::setConfig('short', [
    'className' => 'File',
    'duration' => '+1 hours',
```

(continues on next page)

(continuação da página anterior)

```
    'path' => CACHE,
    'prefix' => 'cake_short_'
]);

// Usando um namespace para nome completo.
Cache::setConfig('long', [
    'className' => 'Cake\Cache\Engine\FileEngine',
    'duration' => '+1 week',
    'probability' => 100,
    'path' => CACHE . 'long' . DS,
]);

// Usando um construtor de objeto
$object = new FileEngine($config);
Cache::setConfig('other', $object);
```

O nome dessas configurações de mecanismo ("curto" e "longo") é usado como o parâmetro `$config` para *Cake\Cache\Cache::write()* e *Cake\Cache\Cache::read()*. Ao configurar mecanismos de cache, você pode consultar o nome da classe usando as seguintes sintaxes:

```
// Nome curto (in App\ or Cake namespaces)
Cache::setConfig('long', ['className' => 'File']);

// Plugin com nome curto
Cache::setConfig('long', ['className' => 'MyPlugin.SuperCache']);

// Namespace completo
Cache::setConfig('long', ['className' => 'Cake\Cache\Engine\FileEngine']);

// Um objeto implementando CacheEngineInterface
Cache::setConfig('long', ['className' => $myCache]);
```

Nota: Ao usar o `FileEngine`, pode ser necessário usar a opção `mask` para garantir que os arquivos de cache sejam criados com as permissões corretas.

Opções do Mecanismo

Cada mecanismo aceita as seguintes opções:

- **`duration` Especifique uma duração padrão para quanto tempo os itens são válidos.**
  Especificada como uma expressão compatível com `strototime()`.
- **`groups` Lista de grupos ou "tags" associados a todas as chaves armazenadas nesta**
  configuração. Útil quando você precisa excluir um subconjunto de dados de um cache.
- **`prefix` Anexado a todas as entradas. É bom para quando você precisar compartilhar**
  um espaço de chave com outra configuração de cache ou outro aplicativo.
- **`probability` Probabilidade de atingir uma limpeza de cache. Definir como 0 o**
  `Cache::gc()` será desativado e não funcionará automaticamente.

Opções do FileEngine

O `FileEngine` usa as seguintes opções específicas do mecanismo:

- `isWindows` Preenchido automaticamente se o host é Windows ou não
- `lock` Os arquivos devem ser bloqueados antes de serem gravados?
- `mask` A máscara usada para arquivos criados
- **`path` Caminho para onde os arquivos de cache devem ser salvos. O padrão** é o diretório temporário do sistema.

Opções RedisEngine

O RedisEngine usa as seguintes opções específicas do mecanismo:

- `port` A porta em que o servidor Redis está sendo executado.
- `host` O host em que o servidor Redis está sendo executado.
- `database` O número do banco de dados a ser usado para conexão.
- `password` Senha do servidor Redis.
- `persistent` Uma conexão persistente deve ser feita com Redis.
- `timeout` Tempo limite de conexão para Redis.
- `unix_socket` Caminho para um soquete unix para Redist.

Opções do MemcacheEngine

- `compress` Se deseja compactar dados.
- `username` Faça login para acessar o servidor Memcache.
- `password` Senha para acessar o servidor Memcache.
- `persistent` O nome da conexão persistente. Todas as configurações que usam o mesmo valor persistente compartilharão uma única conexão subjacente.
- `serialize` O mecanismo do serializador usado para serializar dados. Os mecanismos disponíveis são php, igbinary e json. Ao lado do php, a extensão memcached deve ser compilada com o suporte serializador apropriado.
- `servers` Cadeia ou matriz de servidores com cache de memória. Se for um array, o MemcacheEngine os usará como um pool.
- `options` Opções adicionais para o cliente memcached. Deve ser uma matriz de opção => valor. Use as constantes `\Memcached::OPT_*` como chaves.

Configurando Fallbacks de Cache

No caso de um mecanismo não estar disponível, como o `FileEngine` tentando gravar em uma pasta não gravável ou o `RedisEngine` falhando ao se conectar ao Redis, o mecanismo voltará ao noop `NullEngine` e acionará um erro registrável. Isso impede que o aplicativo lance uma exceção não capturada devido a falha no cache.

Você pode ajustar as configurações de cache para retornar a uma configuração especifica usando a chave de configuração `fallback`:

```
Cache::setConfig('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
    'port' => 6379,
    'fallback' => 'default',
]);
```

Se o servidor Redis falhar inesperadamente, a configuração de cache `redis` retornaria à gravação na configuração de cache `default`. Se a gravação na configuração do cache `default` *também* falhar nesse cenário, o mecanismo retornará novamente ao `NullEngine` e impedirá o aplicativo de lançar uma exceção não capturada.

Você pode desativar fallbacks de cache com `false`:

```
Cache::setConfig('redis', [
    'className' => 'Redis',
    'duration' => '+1 hours',
    'prefix' => 'cake_redis_',
    'host' => '127.0.0.1',
    'port' => 6379,
    'fallback' => false
]);
```

Quando não houver falhas no cache de fallback, serão geradas exceções.

Remoção de Mecanismos de Cache Configurados

static Cake\Cache\Cache::**drop**(*$key*)

Depois que uma configuração é criada, você não pode alterá-la. Em vez disso, você deve descartar a configuração e recriá-la usando *Cake\Cache\Cache::drop()* e *Cake\Cache\Cache::setConfig()*. Descartar um mecanismo de cache removerá a configuração e destruirá o adaptador, se ele tiver sido construído.

Gravando em um Cache

static Cake\Cache\Cache::**write**(*$key*, *$value*, *$config = 'default'*)

`Cache::write()` gravará um $valor no cache. Você pode ler ou excluir esse valor posteriormente consultando-o com `$key`. Você pode especificar uma configuração opcional para armazenar o cache também. Se nenhum `$config` for especificado, o padrão será usado. `Cache::write()` pode armazenar qualquer tipo de objeto e é ideal para armazenar resultados de descobertas de modelos:

```
if (($posts = Cache::read('posts')) === false) {
    $posts = $someService->getAllPosts();
    Cache::write('posts', $posts);
}
```

Usando `Cache::write()` e `Cache::read()` irá reduzir o número de viagens feitas ao banco de dados para buscar postagens.

Nota: Se você planeja armazenar em cache o resultado de consultas feitas com o ORM do CakePHP, é melhor usar os recursos de cache internos do objeto Query, conforme descrito na seção *Cache de Resultados Carregados*

Escrevendo Várias Chaves de uma só Vez

static Cake\Cache\Cache::**writeMany**(*$data*, *$config = 'default'*)

Você pode precisar escrever várias chaves de cache de uma só vez. Embora você possa usar várias chamadas para `write()`, `writeMany()` permite que o CakePHP use APIs de armazenamento mais eficientes, quando disponíveis. Por exemplo, usando `writeMany()` salve várias conexões de rede ao usar o Memcached:

```
$result = Cache::writeMany([
    'article-' . $slug => $article,
    'article-' . $slug . '-comments' => $comments
]);

// $result poderá conter
['article-first-post' => true, 'article-first-post-comments' => true]
```

Armazenamento em Cache de Leitura

static Cake\Cache\Cache::**remember**(*$key*, *$callable*, *$config = 'default'*)

Esse recurso facilita o armazenamento em cache de leitura. Se a chave de cache nomeada existir, ela será retornada. Se a chave não existir, a chamada será invocada e os resultados armazenados no cache da chave fornecida.

Por exemplo, você desejará armazenar em cache os resultados de chamadas de serviço remoto. Você pode usar `remember()` para simplificar:

```
class IssueService
{
    public function allIssues($repo)
    {
        return Cache::remember($repo . '-issues', function () use ($repo) {
            return $this->fetchAll($repo);
        });
    }
}
```

Lendo de um Cache

static Cake\Cache\Cache::**read**(*$key*, *$config = 'default'*)

`Cache::read()` é usado para ler o valor em cache armazenado em `$key` do `$config`. Se `$config` for nulo, a configuração padrão será usada. `Cache::read()` retornará o valor em cache se for um cache válido ou `false` se o cache expirou ou não existe. O conteúdo do cache pode ser avaliado como falso, portanto, use os operadores de comparação estritos: `===` ou `!==`.

Por exemplo:

```
$cloud = Cache::read('cloud');
if ($cloud !== false) {
    return $cloud;
}

// Gere dados na nuvem
// ...

// Armazenar dados no cache
Cache::write('cloud', $cloud);

return $cloud;
```

Ou, se você estiver usando outra configuração de cache chamada `short`, poderá especificá-la nas chamadas `Cache::read()` e `Cache::write()`, conforme abaixo:

```
// Leia a chave "cloud", mas a partir da configuração curta em vez do padrão

$cloud = Cache::read('cloud', 'short');
if ($cloud !== false) {
    return $cloud;
}

// Gere dados na nuvem
// ...

// Armazene dados no cache, usando a configuração de cache "short" em vez do padrão
Cache::write('cloud', $cloud, 'short');

return $cloud;
```

Lendo Várias Chaves de uma só Vez

static Cake\Cache\Cache::**readMany**(*$keys*, *$config = 'default'*)

Depois de escrever várias chaves ao mesmo tempo, você provavelmente também as lerá. Embora você possa usar várias chamadas para `read()`, `readMany()` permite que o CakePHP use APIs de armazenamento mais eficientes, quando disponíveis. Por exemplo, usando `readMany()` salve várias conexões de rede ao usar o Memcached:

```
$result = Cache::readMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
```

(continues on next page)

(continuação da página anterior)

```
]);
// $result poderá conter
['article-first-post' => '...', 'article-first-post-comments' => '...']
```

Exclusão de um Cache

static Cake\Cache\Cache::**delete**(*$key*, *$config = 'default'*)

`Cache::delete()` permitirá remover completamente um objeto em cache da loja:

```
// Remove uma chave
Cache::delete('my_key');
```

Exclusão de Várias Chaves de uma só Vez

static Cake\Cache\Cache::**deleteMany**(*$keys*, *$config = 'default'*)

Depois de escrever várias chaves de uma vez, você pode excluí-las. Embora você possa usar várias chamadas para `delete()`, `deleteMany()` permite que o CakePHP use APIs de armazenamento mais eficientes, quando disponíveis. Por exemplo, usando `deleteMany()` remove várias conexões de rede ao usar o Memcached:

```
$result = Cache::deleteMany([
    'article-' . $slug,
    'article-' . $slug . '-comments'
]);
// $result conterá
['article-first-post' => true, 'article-first-post-comments' => true]
```

Limpando Dados em Cache

static Cake\Cache\Cache::**clear**(*$check*, *$config = 'default'*)

Destrua todos os valores em cache para uma configuração de cache. Em mecanismos como: Apcu, Memcached e Wincache, o prefixo da configuração do cache é usado para remover as entradas do cache. Verifique se diferentes configurações de cache têm prefixos diferentes:

```
// Limpa apenas as chaves expiradas.
Cache::clear(true);

// Limpará todas as chaves.
Cache::clear(false);
```

Nota: Como o APCu e o Wincache usam caches isolados para servidor da web e CLI, eles devem ser limpos separadamente (a CLI não pode limpar o servidor da web e vice-versa).

Usando Cache para Armazenar Contadores

static Cake\Cache\Cache::**increment**(*$key*, *$offset = 1*, *$config = 'default'*)

static Cake\Cache\Cache::**decrement**(*$key*, *$offset = 1*, *$config = 'default'*)

Os contadores no seu aplicativo são bons candidatos para armazenamento em cache. Como exemplo, uma contagem regressiva simples para os "slots" restantes em uma disputa pode ser armazenada no cache. A classe Cache expõe maneiras atômicas de aumentar/diminuir os valores dos contadores de maneira fácil. As operações atômicas são importantes para esses valores, pois reduzem o risco de contenção e a capacidade de dois usuários reduzirem simultaneamente o valor em um, resultando em um valor incorreto.

Depois de definir um valor inteiro, você pode manipulá-lo usando `increment()` e `decrement()`:

```
Cache::write('initial_count', 10);

// Decrementa
Cache::decrement('initial_count');

// Ou
Cache::increment('initial_count');
```

Nota: Incrementar e decrementar não funcionam com o `FileEngine`. Você deve usar APCu, Wincache, Redis ou Memcached.

Usando o Cache para Armazenar Resultados Comuns de Consulta

Você pode melhorar bastante o desempenho do seu aplicativo colocando resultados que raramente mudam ou estão sujeitos a leituras pesadas no cache. Um exemplo perfeito disso são os resultados de *Cake\ORM\Table::find()*. O objeto Query permite armazenar resultados em cache usando o método `cache()`. Veja a seção *Cache de Resultados Carregados* para mais informações.

Usando Grupos

Às vezes, você deseja marcar várias entradas de cache para pertencer a determinado grupo ou namespace. Esse é um requisito comum para chaves de invalidação em massa sempre que algumas informações são alteradas e são compartilhadas entre todas as entradas no mesmo grupo. Isso é possível declarando os grupos na configuração de cache:

```
Cache::setConfig('site_home', [
    'className' => 'Redis',
    'duration' => '+999 days',
    'groups' => ['comment', 'article']
]);
```

Cake\Cache\Cache::**clearGroup**(*$group*, *$config = 'default'*)

Digamos que você deseja armazenar o HTML gerado para sua página inicial no cache, mas também deseja invalidá-lo automaticamente sempre que um comentário ou postagem for adicionado ao seu banco de dados. Adicionando os grupos `comment` e `article`, identificamos efetivamente qualquer chave armazenada nessa configuração de cache com os dois nomes de grupos.

Por exemplo, sempre que uma nova postagem é adicionada, poderíamos dizer ao mecanismo de cache para remover todas as entradas associadas ao grupo `article`:

```
// src/Model/Table/ArticlesTable.php
public function afterSave($event, $entity, $options = [])
{
    if ($entity->isNew()) {
        Cache::clearGroup('article', 'site_home');
    }
}
```

static Cake\Cache\Cache::**groupConfigs**(*$group = null*)

`groupConfigs()` pode ser usado para recuperar o mapeamento entre o grupo e as configurações, ou seja: ter o mesmo grupo:

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

/**
 * Uma variação do exemplo anterior que limpa todas as
 * configurações de cache com o mesmo grupo
 */
public function afterSave($event, $entity, $options = [])
{
    if ($entity->isNew()) {
        $configs = Cache::groupConfigs('article');
        foreach ($configs['article'] as $config) {
            Cache::clearGroup('article', $config);
        }
    }
}
```

Os grupos são compartilhados em todas as configurações de cache usando o mesmo mecanismo e o mesmo prefixo. Se você estiver usando grupos e quiser tirar proveito da exclusão do grupo, escolha um prefixo comum para todas as suas configurações.

Ativar ou Desativar Globalmente o Cache

static Cake\Cache\Cache::**disable**

Pode ser necessário desativar todas as leituras e gravações do cache ao tentar descobrir problemas relacionados à expiração do cache. Você pode fazer isso usando `enable()` e `disable()`:

```
// Desative todas as leituras de cache e gravações de cache.
Cache::disable();
```

Uma vez desativado, todas as leituras e gravações retornarão `null`.

static Cake\Cache\Cache::**enable**

Uma vez desativado, você pode usar `enable()` para reativar o cache:

```
// Reative todas as leituras e gravações do cache.
Cache::enable();
```

static Cake\Cache\Cache::**enabled**

Se você precisar verificar o estado do cache, poderá usar `enabled()`.

Criando um Mecanismo de Cache

Você pode fornecer mecanismos personalizados de `Cache` em `App\Cache\Engine`, bem como em plugins usando `$plugin\Cache\Engine`. Os mecanismos de cache devem estar em um diretório de cache. Se você tivesse um mecanismo de cache chamado `MyCustomCacheEngine`, ele seria colocado em **src/Cache/Engine/MyCustomCacheEngine.php**. Ou em **plugins/MyPlugin/src/Cache/Engine/MyCustomCacheEngine.php** como parte de um plug-in. As configurações de cache dos plugins precisam usar a sintaxe de pontos do plug-in:

```
Cache::setConfig('custom', [
    'className' => 'MyPlugin.MyCustomCache',
    // ...
]);
```

Os mecanismos de cache personalizado devem estender *Cake\Cache\CacheEngine*, que define vários métodos abstratos, além de fornecer alguns métodos de inicialização.

A API necessária para um CacheEngine é

class Cake\Cache**CacheEngine**

A classe base para todos os mecanismos de cache usados com o Cache.

Cake\Cache\CacheEngine::**write**(*$key*, *$value*)

Retorno
boolean para sucesso.

Escreva o valor de uma chave no cache, retorna `true` se os dados foram armazenados em cache com sucesso, `false` em caso de falha.

Cake\Cache\CacheEngine::**read**(*$key*)

Retorno
O valor em cache ou `false` para falha.

Leia uma chave do cache. Retorne `false` para indicar que a entrada expirou ou não existe.

Cake\Cache\CacheEngine::**delete**(*$key*)

Retorno
Booleano `true` para sucesso.

Exclua uma chave do cache. Retorne `false` para indicar que a entrada não existia ou não pôde ser excluída.

Cake\Cache\CacheEngine::**clear**(*$check*)

Retorno
Booleano `true` para sucesso.

Exclua todas as chaves do cache. Se $check for `true`, você deve validar se cada valor realmente expirou.

Cake\Cache\CacheEngine::**clearGroup**(*$group*)

Retorno
Booleano `true` para sucesso.

Exclua todas as chaves do cache pertencentes ao mesmo grupo.

Cake\Cache\CacheEngine::**decrement**(*$key*, *$offset = 1*)

Retorno
Booleano `true` para sucesso.

Decrementar um número na chave e retorna o valor decrementado

Cake\Cache\CacheEngine::**increment**(*$key*, *$offset = 1*)

Retorno
Booleano `true` para sucesso.

Incremente um número abaixo da chave e retorna valor incrementado

CAPÍTULO 15

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 vocë 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) constutuída a partir do PHP deve estar disponível no sistema se você planeja utilizr 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 recem criado:

```
namespace App\Shell;

use Cake\Console\Shell;

class HelloShell extends Shell
{
```

(continues on next page)

(continuação da página anterior)

```
    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.');
    }

    public function heyThere($name = 'Anonymous')
    {
        $this->out('Hey there ' . $name);
    }
}
```

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.');
        }
        $user = $this->Users->findByUsername($this->args[0])->first();
        $this->out(print_r($user, true));
    }
}
```

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 *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*, *$defaut* = *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 sobreescrever o arquivo caso já exista. Se a propriedade de interação da shell for `false`, nenhuma questão será disparada e o arquivo será simplesmente sobreescrito.

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 sobreescrever 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 saíde da seguinte forma:

```
// Deve aparecer em todos os níveis.
$this->out('Quiet message', 1, Shell::QUIET);
```

(continues on next page)

(continuação da página anterior)

```
$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 saíde 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 seqüência 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 saíde você pode fazer:

```
$this->_io->styles('flashy', ['text' => 'magenta', 'blink' => true]);
```

Isso deve então permití-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>Whoooa</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 boleanos, 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, haverão ocasiões que você quererá 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 interpetador shell ao chamar `$this->getOptionParser()`. Sobreescrever esse método permite configurar o OptionParser para definir as entradas aguardadas da sua shell. Você também pode configurar interpetadores 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 indefinodo, 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, suprindo padrões e criando ativadores boleanos. 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 boleanos. Estes ativadores não consumem 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 boleano. 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 boleanas

As opções podem ser definidas como opções boleanas, que são úteis quando você precisa criar algumas opções de marcação. Como opções com padrões, opções boleanas 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 boleanas:

```
if ($this->params['verbose']) {
    // Do something.
}
```

Desde que as opções boleanas 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 específios. 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],
```

(continues on next page)

(continuação da página anterior)

```
            '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">
```

(continues on next page)

(continuação da página anterior)

```
            <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 atua 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, sinta-se 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.

Console Interativo (REPL)

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[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.

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␣
```

(continues on next page)

[97] https://github.com/cakephp/docs
[98] https://github.com/cakephp/docs

(continuação da página anterior)

```
↪os nomes)
# |     |     |     \————————— mês (1 - 12)
# |     |     \———————————————— 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, 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.

Completion 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[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.

Plugin 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[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.

---

[99] https://github.com/cakephp/docs
[100] https://github.com/cakephp/docs
[101] https://github.com/cakephp/docs

Routes 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[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.

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[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.

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
```

[102] https://github.com/cakephp/docs
[103] https://github.com/cakephp/docs

CAPÍTULO 16

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()

Se você tem o Psysh[104] 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.

[104] https://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 valido 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.

CAPÍTULO 17

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 [
```

(continues on next page)

(continuação da página anterior)

```
    '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 *Falsificação de Solicitação entre Sites (CSRF) Middleware*.
- Você pode querer habilitar o *Security (Segurança)*. 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
```

CAPÍTULO 18

Email

class Cake\Mailer**Mailer**(*string|array|null $profile = null*)

`Mailer` é uma classe conveniente para enviar e-mail. Com esta aula você pode enviar e-mail de qualquer lugar do seu aplicativo.

Uso Básico

Em primeiro lugar, você deve garantir que a classe seja carregada:

```
use Cake\Mailer\Mailer;
```

Depois de carregar o `Mailer`, você pode enviar um e-mail com o seguinte código:

```
$mailer = new Mailer('default');
$mailer->setFrom(['me@example.com' => 'My Site'])
    ->setTo('you@example.com')
    ->setSubject('About')
    ->deliver('My message');
```

Como os métodos setter do `Mailer` retornam a instância da classe, você pode definir suas propriedades com encadeamento de métodos.

`Mailer` tem vários métodos para definir destinatários - `setTo()`, `setCc()`, `setBcc()`, `addTo()`, `addCc()` e `addBcc()`. A principal diferença é que os três primeiros sobrescreverão o que já foi definido e o último apenas adicionará mais destinatários aos seus respectivos campos:

```
$mailer = new Mailer();
$mailer->setTo('to@example.com', 'To Example');
$mailer->addTo('to2@example.com', 'To2 Example');
```

(continues on next page)

(continuação da página anterior)

```
// Os destinatários do e-mail são: to@example.com e to2@example.com
$mailer->setTo('test@example.com', 'ToTest Example');
// O destinatário do e-mail é: test@example.com
```

Escolha do Remetente

Ao enviar e-mail em nome de outras pessoas, geralmente é uma boa ideia definir o remetente original usando o cabeçalho Sender. Você pode fazer isso usando `setSender()`:

```
$mailer = new Mailer();
$mailer->setSender('app@example.com', 'MyApp emailer');
```

Nota: Também é uma boa ideia definir o remetente ao enviar o e-mail em nome de outra pessoa. Isso os impede de receber qualquer mensagem sobre a situação da entrega.

Configuração

Perfis de Email e configurações de transporte de e-mail são definidos nos arquivos de configuração do seu aplicativo. As chaves `Email` e `EmailTransport` definem perfis de mailer e configurações de transporte de e-mail, respectivamente. Durante a inicialização do aplicativo, os ajustes de configuração são passados de `Configure` para as classes `Mailer` e `TransportFactory` usando `setConfig()`. Ao definir perfis e transportes, você pode manter o código do aplicativo livre de dados de configuração e evitar a duplicação o que torna a manutenção e a implantação mais difíceis.

Para carregar uma configuração predefinida, você pode usar o método `setProfile()` ou passá-lo para o construtor do `Mailer`:

```
$mailer = new Mailer();
$mailer->setProfile('default');

// Ou no construtor
$mailer = new Mailer('default');
```

Em vez de passar uma string que corresponda a um nome de configuração predefinido, você pode simplesmente carregar um conjunto de opções:

```
$mailer = new Mailer();
$mailer->setProfile(['from' => 'me@example.org', 'transport' => 'my_custom']);

// Ou no construtor
$mailer = new Mailer(['from' => 'me@example.org', 'transport' => 'my_custom']);
```

Perfis de Configuração

Definir perfis de entrega permite que você consolide configurações de e-mail comuns em perfis reutilizáveis. Seu aplicativo pode ter quantos perfis forem necessários. As seguintes chaves de configuração são usadas:

- `'from'`: Mailer ou matriz de remetente. Veja `Mailer::setFrom()`.
- `'sender'`: Mailer ou matriz do remetente real. Veja `Mailer::setSender()`.
- `'to'`: Mailer ou matriz de destinos. Veja `Mailer::setTo()`.
- `'cc'`: Mailer ou matriz de cópias. Veja `Mailer::setCc()`.
- `'bcc'`: Mailer ou matriz de cópias ocultas. Veja `Mailer::setBcc()`.
- `'replyTo'`: Mailer ou matriz para resposta de e-mail. Veja `Mailer::setReplyTo()`.
- `'readReceipt'`: Endereço de correspondência ou uma série de endereços para receber a confirmação de leitura. Veja `Mailer::setReadReceipt()`.
- `'returnPath'`: Endereço do mailer ou uma série de endereços a serem retornados se houver algum erro. Veja `Mailer::setReturnPath()`.
- `'messageId'`: ID da mensagem de e-mail. Veja `Mailer::setMessageId()`.
- `'subject'`: Assunto da mensagem. Veja `Mailer::setSubject()`.
- `'message'`: Conteúdo da mensagem. Não defina este campo se estiver usando conteúdo renderizado.
- `'priority'`: Prioridade do e-mail como valor numérico (geralmente de 1 a 5, sendo 1 o mais alto).
- `'headers'`: Cabeçalhos a serem incluídos. Veja `Mailer::setHeaders()`.
- `'viewRender'`: Se você estiver usando conteúdo renderizado, defina o nome da classe da visualização. Veja `Mailer::viewRender()`.
- `'template'`: Se você estiver usando conteúdo renderizado, defina o nome do template. Veja `ViewBuilder::setTemplate()`.
- `'theme'`: Tema usado ao renderizar o template. Veja `ViewBuilder::setTheme()`.
- `'layout'`: Se você estiver usando conteúdo renderizado, defina o layout para renderizar. Se você deseja renderizar um template sem layout, defina este campo como nulo. Veja `ViewBuilder::setTemplate()`.
- `'viewVars'`: Se você estiver usando conteúdo renderizado, defina a matriz com variáveis a serem usadas na visualização. Veja `Mailer::setViewVars()`.
- `'attachments'`: Lista de arquivos para anexar. Veja `Mailer::setAttachments()`.
- `'emailFormat'`: Formato de email (html, text ou ambos). Veja `Mailer::setEmailFormat()`.
- `'transport'`: Nome da configuração de Transporte. Veja *Configurando os Transportes*.
- `'log'`: Nível de logs para registrar os cabeçalhos e a mensagem do e-mail. `true` usará LOG_DEBUG. Veja *logging-levels*. Observe que os logs serão emitidos sob o escopo chamado `email`. Veja também *logging-scopes*.
- `'helpers'`: Conjunto de auxiliares usados no template de e-mail. `ViewBuilder::setHelpers()`.

Nota: Os valores das chaves acima usando Mailer ou matriz, como from, to, cc, etc, serão passados como o primeiro parâmetro dos métodos correspondentes. O equivalente para: `$mailer->setFrom('my@example.com', 'My Site')` seria definido como `'from' => ['my@example.com' => 'My Site']` em sua configuração

Configurando Cabeçalhos

No `Mailer` você é livre para definir os cabeçalhos que quiser. Não se esqueça de colocar o prefixo `X-` para seus cabeçalhos personalizados.

Veja `Mailer::setHeaders()` e `Mailer::addHeaders()`

Envio de Emails com Templates

Muitas vezes, os emails são muito mais do que uma simples mensagem de texto. Para facilitar isso, o CakePHP fornece uma maneira de enviar emails usando *view layer* do CakePHP.

Os modelos para emails residem em uma pasta especial `templates/email` de seu aplicativo. As visualizações do mailer também podem usar layouts e elementos como templates normais:

```
$mailer = new Mailer();
$mailer
            ->setEmailFormat('html')
            ->setTo('bob@example.com')
            ->setFrom('app@domain.com')
            ->viewBuilder()
                ->setTemplate('welcome')
                ->setLayout('fancy');

$mailer->deliver();
```

O exemplo acima usaria **templates/email/html/welcome.php** para a visualização e **templates/layout/email/html/fancy.php** para o layout. Você também pode enviar mensagens de e-mail com modelo de várias partes:

```
$mailer = new Mailer();
$mailer
            ->setEmailFormat('both')
            ->setTo('bob@example.com')
            ->setFrom('app@domain.com')
            ->viewBuilder()
                ->setTemplate('welcome')
                ->setLayout('fancy');

$mailer->deliver();
```

Isso usaria os seguintes arquivos de modelo:

- **templates/email/text/welcome.php**
- **templates/layout/email/text/fancy.php**
- **templates/email/html/welcome.php**
- **templates/layout/email/html/fancy.php**

Ao enviar e-mails com modelo, você tem a opção de enviar `text`, `html` ou `both`.

Você pode definir todas as configurações relacionadas ao template usando a instância do construtor de views obtida por `Mailer::viewBuilder()` semelhante como você faz o no controlador.

Você pode definir variáveis de visualização com `Mailer::setViewVars()`:

```
$mailer = new Mailer('templated');
$mailer->setViewVars(['value' => 12345]);
```

Ou você pode usar os métodos construtores de visualização `ViewBuilder::setVar()` e `ViewBuilder::setVars()`.

Em seus modelos de e-mail, você pode usá-los com:

```
<p>Aqui está o seu valor: <b><?= $value ?></b></p>
```

Você também pode usar ajudantes em e-mails, da mesma forma que em arquivos de modelo normais. Por padrão, apenas o `HtmlHelper` é carregado. Você pode carregar auxiliares adicionais usando o método `ViewBuilder::setHelpers()`:

```
$mailer->viewBuilder()->setHelpers(['Html', 'Custom', 'Text']);
```

Ao configurar ajudantes, certifique-se de incluir "Html" ou ele será removido dos ajudantes carregados em seu modelo de email.

Se você deseja enviar e-mail usando templates em um plugin, você pode usar a familiar *sintaxe plugin* para fazer isso:

```
$mailer = new Mailer();
$mailer->viewBuilder()->setTemplate('Blog.new_comment');
```

O exemplo acima usaria o template e o layout do plugin do Blog.

Em alguns casos, pode ser necessário substituir o modelo padrão fornecido pelos plug-ins. Você pode fazer isso usando temas:

```
$mailer->viewBuilder()
    ->setTemplate('Blog.new_comment')
    ->setLayout('Blog.auto_message')
    ->setTheme('TestTheme');
```

Isso permite que você sobrescreva o template `new_comment` em seu tema sem modificar o plugin do Blog. O arquivo de modelo deve ser criado no seguinte caminho: **templates/plugin/TestTheme/plugin/Blog/email/text/new_comment.php**.

Enviando Anexos

`Cake\Mailer\Mailer::setAttachments`(*$attachments*)

Você também pode anexar arquivos a mensagens de e-mail. Existem alguns formatos diferentes, dependendo do tipo de arquivo que você possui e de como deseja que os nomes dos arquivos apareçam no cliente de e-mail do destinatário:

1. Matriz: `$mailer->setAttachments(['/full/file/path/file.png'])` terá o mesmo comportamento de usar uma string.
2. Matriz com chave: `$mailer->setAttachments(['photo.png' => '/full/some_hash.png'])` irá anexar some_hash.png com o nome photo.png. O destinatário verá photo.png, não some_hash.png.
3. Matrizes aninhadas:

```
$mailer->setAttachments([
    'photo.png' => [
        'file' => '/full/some_hash.png',
```

(continues on next page)

(continuação da página anterior)

```
        'mimetype' => 'image/png',
        'contentId' => 'my-unique-id'
    ]
]);
```

O código acima anexará o arquivo com um tipo MIME diferente e com Content ID personalizado (ao definir o ID de conteúdo, o anexo é transformado em inline). O mimetype e contentId são opcionais neste formulário.

3.1. Quando você está usando o `contentId`, você pode usar o arquivo no corpo do HTML como `<img src="cid:my-content-id">`.

3.2. Você pode usar a opção `contentDisposition` para desabilitar o cabeçalho `Content-Disposition` para um anexo. Isso é útil ao enviar convites ical a clientes usando o Outlook.

3.3 Em vez da opção `file`, você pode fornecer o conteúdo do arquivo como uma string usando a opção `data`. Isso permite que você anexe arquivos sem a necessidade de caminhos de arquivo para eles.

Regras para Validação de Endereço mais Flexíveis

`Cake\Mailer\Mailer::setEmailPattern`(*$pattern*)

Se estiver tendo problemas de validação ao enviar para endereços não compatíveis, você pode relaxar o padrão usado para validar endereços de e-mail. Isso às vezes é necessário ao lidar com alguns ISP's:

```
$mailer = new Mailer('default');

// Relaxe o padrão de e-mail,
// para que você possa enviar para endereços não conformes.
$mailer->setEmailPattern($newPattern);
```

Enviando Mensagens Rapidamente

Às vezes, você precisa de uma maneira rápida para enviar um e-mail e não quer necessariamente definir várias configurações com antecedência. `Cake\Mailer\Email::deliver()` é destinado a esse propósito.

Você pode criar sua configuração usando `Cake\Mailer\Email::config()`, ou usar uma matriz com todas as opções que você precisa e usar o método estático `Email::deliver()`. Exemplo:

```
Email::deliver('you@example.com', 'Subject', 'Message', ['from' => 'me@example.com']);
```

Este método enviará um e-mail para «you@example.com», de «me@example.com» com assunto «Subject» e conteúdo «Message».

O retorno de `deliver()` é uma instância `Cake\Mailer\Email` com todas as configurações definidas. Se você não deseja enviar o e-mail imediatamente e deseja configurar algumas coisas antes de enviar, pode passar o quinto parâmetro como `false`.

O terceiro parâmetro é o conteúdo da mensagem ou uma matriz com variáveis (ao usar conteúdo renderizado).

O 4º parâmetro pode ser um array com as configurações ou uma string com o nome da configuração em `Configure`.

Se você quiser, pode passar o destinatário, o assunto e a mensagem como nulos e fazer todas as configurações no 4º parâmetro (como array ou usando `Configure`). Verifique a lista de *configurations* para ver todas as configurações aceitas.

Enviando E-mails de CLI

Ao enviar emails em um script CLI (Shells, Tasks, ...), você deve definir manualmente o nome de domínio a ser usado pelo Mailer. Ele servirá como o nome do host para o id da mensagem (uma vez que não há nome de host em um ambiente CLI):

```
$mailer->setDomain('www.example.org');
// Resultados em ids de mensagens como ``<UUID@www.example.org>`` (válido)
// Ao invés de `<UUID@>`` (inválido)
```

Um id de mensagem válido pode ajudar a evitar que emails acabem em pastas de spam.

Criação de Emails Reutilizáveis

Até agora vimos como usar diretamente a classe `Mailer` para criar e enviar emails. Mas a principal característica do mailer é permitir a criação de emails reutilizáveis em todo o seu aplicativo. Eles também podem ser usados para conter várias configurações de e-mail em um local. Isso ajuda a manter seu código DRYer e mantém os ruídos de configuração de e-mail longe de outras áreas do seu aplicativo.

Neste exemplo, estaremos criando um `Mailer` que contém emails relacionados ao usuário. Para criar nosso `UserMailer`, crie o arquivo **src/Mailer/UserMailer.php**. O conteúdo do arquivo deve ser semelhante ao seguinte:

```
namespace App\Mailer;

use Cake\Mailer\Mailer;

class UserMailer extends Mailer
{
    public function welcome($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject(sprintf('Welcome %s', $user->name))
            ->viewBuilder()
                ->setTemplate('welcome_mail'); // Por padrão, é usado um modelo com o␣
↪mesmo nome do método.
    }

    public function resetPassword($user)
    {
        $this
            ->setTo($user->email)
            ->setSubject('Reset password')
            ->setViewVars(['token' => $user->token]);
    }
}
```

Em nosso exemplo, criamos dois métodos, um para enviar um e-mail de boas-vindas e outro para enviar um e-mail de redefinição de senha. Cada um desses métodos espera um usuário `Entity` e utiliza suas propriedades para configurar cada e-mail.

Agora podemos usar nosso `UserMailer` para enviar nossos e-mails relacionados ao usuário de qualquer lugar em nosso aplicativo. Por exemplo, se quisermos enviar nosso e-mail de boas-vindas, poderíamos fazer o seguinte:

```
namespace App\Controller;

use Cake\Mailer\MailerAwareTrait;

class UsersController extends AppController
{
    use MailerAwareTrait;

    public function register()
    {
        $user = $this->Users->newEmptyEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData())
            if ($this->Users->save($user)) {
                $this->getMailer('User')->send('welcome', [$user]);
            }
        }
        $this->set('user', $user);
    }
}
```

Se quisermos separar completamente o envio de um e-mail de boas-vindas ao usuário do código de nosso aplicativo, podemos fazer com que nosso `UserMailer` se inscreva no evento `Model.afterSave`. Ao inscrever-se em um evento, podemos manter as classes relacionadas ao usuário de nosso aplicativo completamente livres de lógica e instruções relacionadas a email. Por exemplo, poderíamos adicionar o seguinte ao nosso `UserMailer`:

```
public function implementedEvents()
{
    return [
        'Model.afterSave' => 'onRegistration'
    ];
}

public function onRegistration(EventInterface $event, EntityInterface $entity,␣
↪ArrayObject $options)
{
    if ($entity->isNew()) {
        $this->send('welcome', [$entity]);
    }
}
```

Agora você pode registrar o mailer como um ouvinte de evento e o método `onRegistration()` será invocado toda vez que o evento `Model.afterSave` for disparado:

```
// anexar ao gerenciador de eventos de usuários
$this->Users->getEventManager()->on($this->getMailer('User'));
```

Nota: Para informações sobre como registrar objetos ouvintes de eventos, por favor consulte a documentação *Registrando Listeners*.

Configurando os Transportes

As mensagens de e-mail são entregues por transportes. Transportes diferentes permitem que você envie mensagens via função `mail()` do PHP, servidores SMTP ou o que for mais útil para depuração. A configuração de transportes permite que você mantenha os dados de configuração fora do código do aplicativo e torna a implantação mais simples, pois você pode simplesmente alterar os dados de configuração. Um exemplo de configuração de transporte se parece com isso:

```
// In config/app.php
'EmailTransport' => [
    // Configuração de amostra de Email
    'default' => [
        'className' => 'Mail',
    ],
    // Amostra de configuração SMTP
    'gmail' => [
        'host' => 'smtp.gmail.com',
        'port' => 587,
        'username' => 'my@gmail.com',
        'password' => 'secret',
        'className' => 'Smtp',
        'tls' => true
    ]
],
```

Os transportes também podem ser configurados em tempo de execução usando `TransportFactory::setConfig()`:

```
use Cake\Mailer\TransportFactory;

// Define um transporte STMP
TransportFactory::setConfig('gmail', [
    'host' => 'ssl://smtp.gmail.com',
    'port' => 465,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp'
]);
```

Você pode configurar servidores SSL SMTP, como Gmail. Para fazer isso, coloque o prefixo `ssl://` no host e configure o valor da porta de acordo. Você também pode habilitar TLS SMTP usando a opção `tls`:

```
use Cake\Mailer\TransportFactory;

TransportFactory::setConfig('gmail', [
    'host' => 'smtp.gmail.com',
    'port' => 587,
    'username' => 'my@gmail.com',
    'password' => 'secret',
    'className' => 'Smtp',
    'tls' => true
]);
```

A configuração acima permitiria a comunicação TLS para mensagens de e-mail.

Para configurar seu mailer para usar um transporte específico, você pode usar o método `Cake\Mailer\Mailer::setTransport()` ou ter o transporte em sua configuração:

```
// Use um transporte nomeado já configurado usando TransportFactory::setConfig()
$mailer->setTransport('gmail');

// Use um objeto construído.
$mailer->setTransport(new \Cake\Mailer\Transport\DebugTransport());
```

Aviso: You will need to have access for less secure apps enabled in your Google account for this to work:

Você precisará ter acesso a aplicativos menos seguros ativados em sua conta do Google para que isso funcione: Permitir que aplicativos menos seguros acessem sua conta[105].

Nota: Configurções STMP do Gmail[106].

Nota: Para usar SSL + SMTP, você precisará ter o SSL configurado na instalação do PHP.

As opções de configuração também podem ser fornecidas como uma string *DSN*. Isso é útil ao trabalhar com variáveis de ambiente ou provedores *PaaS*:

```
TransportFactory::setConfig('default', [
    'url' => 'smtp://my@gmail.com:secret@smtp.gmail.com:587?tls=true',
]);
```

Ao usar uma string DSN, você pode definir quaisquer parâmetros/opções adicionais como argumentos de string de consulta.

static Cake\Mailer\Mailer::**drop**(*$key*)

Depois de configurados, os transportes não podem ser modificados. Para modificar um transporte, você deve primeiro descartá-lo e reconfigurá-lo.

Criação de Transportes Personalizados

Você pode criar seus transportes personalizados para, por exemplo, enviar e-mail usando serviços como SendGrid, MailGun, Postmark etc. Para criar seu transporte, primeiro crie o arquivo **src/Mailer/Transport/ExampleTransport.php** (onde Exemple é o nome do seu transporte). Para começar, seu arquivo deve ser semelhante a:

```
namespace App\Mailer\Transport;

use Cake\Mailer\AbstractTransport;
use Cake\Mailer\Message;

class ExampleTransport extends AbstractTransport
{
```

(continues on next page)

[105] https://support.google.com/accounts/answer/6010255
[106] https://support.google.com/a/answer/176600?hl=en

(continuação da página anterior)

```
    public function send(Message $message): array
    {
        // Faça alguma coisa.
    }
}
```

Você deve implementar o método `send(Mailer $mailer)` com sua lógica personalizada.

Enviar Emails sem Usar o Mailer

O `Mailer` é uma classe de abstração de nível superior que atua como uma ponte entre as classes `Cake\Mailer\Message`, `Cake\Mailer\Renderer` e `Cake\Mailer\AbstractTransport` para facilitar a configuração e entrega do e-mail.

Se você quiser, pode usar essas classes diretamente com o `Mailer` também.

Por exemplo:

```
$render = new \Cake\Mailer\Renderer();
$render->viewBuilder()
    ->setTemplate('custom')
    ->setLayout('sparkly');

$message = new \Cake\Mailer\Message();
$message
    ->setFrom('admin@cakephp.org')
    ->setTo('user@foo.com')
    ->setBody($render->render());

$transport = new \Cake\Mailer\Transport\MailTransport();
$result = $transport->send($message);
```

Você pode até pular usando o `Renderer` e definir o corpo da mensagem diretamente usando os métodos `Message::setBodyText()` e `Message::setBodyHtml()`.

Testando Emails

Para testar os mailers, adicione `Cake\TestSuite\EmailTrait` ao seu caso de teste. O `MailerTrait` usa ganchos PHPUnit para substituir os transportes de e-mail de sua aplicação por um proxy que intercepta mensagens de e-mail e permite que você faça afirmações sobre o e-mail que será entregue.

Adicione a trait ao seu caso de teste para começar a testar e-mails e carregar rotas se seus e-mails precisarem gerar URLs:

```
namespace App\Test\TestCase\Mailer;

use App\Mailer\WelcomeMailer;
use App\Model\Entity\User;

use Cake\TestSuite\EmailTrait;
```

(continues on next page)

(continuação da página anterior)

```
use Cake\TestSuite\TestCase;

class WelcomeMailerTestCase extends TestCase
{
    use EmailTrait;

    public function setUp(): void
    {
        parent::setUp();
        $this->loadRoutes();
    }
}
```

Vamos supor que temos um mailer que entrega e-mails de boas-vindas quando um novo usuário se registra. Queremos verificar se o assunto e o corpo contêm o nome do usuário:

```
// em sua classe WelcomeMailerTestCase.
public function testName()
{
    $user = new User([
        'name' => 'Alice Alittea',
        'email' => 'alice@example.org',
    ]);
    $mailer = new WelcomeMailer();
    $mailer->send('welcome', [$user]);

    $this->assertMailSentTo($user->email);
    $this->assertMailContainsText('Hi ' . $user->name);
    $this->assertMailContainsText('Welcome to CakePHP!');
}
```

Métodos de Asserções

A trait `Cake\TestSuite\EmailTrait` fornece as seguintes asserções:

```
// Afirma que um número esperado de e-mails foi enviado
$this->assertMailCount($count);

// Afirma que nenhum e-mail foi enviado
$this->assertNoMailSent();

// Afirma que um e-mail foi enviado para um endereço
$this->assertMailSentTo($address);

// Afirma que um e-mail foi enviado de um endereço
$this->assertMailSentFrom($address);

// Afirma que um e-mail contém o conteúdo esperado
$this->assertMailContains($contents);

// Afirma que um e-mail contém conteúdo html esperado
```

(continues on next page)

(continuação da página anterior)

```
$this->assertMailContainsHtml($contents);

// Afirma que um e-mail contém o conteúdo de texto esperado
$this->assertMailContainsText($contents);

// Afirma que um e-mail contém o valor esperado em um getter de mensagem (por exemplo,
↪"assunto")
$this->assertMailSentWith($expected, $parameter);

// Afirma que um e-mail em um índice específico foi enviado para um endereço
$this->assertMailSentToAt($at, $address);

// Afirma que um e-mail em um índice específico foi enviado de um endereço
$this->assertMailSentFromAt($at, $address);

// Afirma que um e-mail em um índice específico contém o conteúdo esperado
$this->assertMailContainsAt($at, $contents);

// Afirma que um e-mail em um índice específico contém o conteúdo html esperado
$this->assertMailContainsHtmlAt($at, $contents);

// Afirma um e-mail em um índice específico contém o conteúdo de texto esperado
$this->assertMailContainsTextAt($at, $contents);

// Afirma que um e-mail contém um anexo
$this->assertMailContainsAttachment('test.png');

// Afirma que um e-mail em um índice específico contém o valor esperado em um getter de␣
↪mensagem (por exemplo, "assunto")
$this->assertMailSentWithAt($at, $expected, $parameter);
```

CAPÍTULO 19

Erros & Exceções

Os aplicativos CakePHP vêm com a configuração de tratamento de erros e exceções para você. Os erros do PHP são capturados e exibidos ou registrados. Exceções não capturadas são renderizadas em páginas de erro automaticamente.

Configurações de Erro & Exceções

A configuração do erro é feita no arquivo **config/app.php** do seu aplicativo. Por padrão, o CakePHP usa `Cake\Error\ErrorHandler` para lidar com erros e exceções do PHP por padrão. A configuração de erro permite personalizar o tratamento de erros para o seu aplicativo. As seguintes opções são suportadas:

- `errorLevel` - int - O nível de erros que você está interessado em capturar. Use as constantes de erro embutidas no PHP e máscaras de bits para selecionar o nível de erro no qual você precisa. Você pode configurá-lo como `E_ALL ^ E_USER_DEPRECATED` para desativar os avisos de depreciação.
- `trace` - bool - Inclua rastreamentos de pilha para erros nos arquivos de log. Rastreamentos de pilha serão incluídos no log após cada erro. Isso é útil para descobrir onde/quando os erros estão sendo gerados
- `exceptionRenderer` - string - A classe responsável por renderizar exceções não capturadas. Se você escolher uma classe personalizada, coloque o arquivo dessa classe em **src/Error**. Esta classe precisa implementar o método `render()`.
- `log` - bool - Quando `true`, as exceções + seus rastreamentos de pilha serão registrados em `Cake\Log\Log`
- `skipLog` - array - Uma matriz de nomes de classes de exceção que não devem ser registrados. Isso é útil para remover NotFoundExceptions ou outras mensagens de log comuns, mas desinteressantes.
- `extraFatalErrorMemory` - int - Defina como o número de megabytes para aumentar o limite de memória quando um erro fatal for encontrado. Isso permite que o espaço sobrando complete o registro ou o tratamento de erros.

Por padrão, os erros do PHP são exibidos quando `debug` é `true` e registrados quando o debug é `false`. O manipulador de erro fatal será chamado independente da configuração do nível `debug` ou `errorLevel`, mas o resultado será diferente

com base no nível de debug. O comportamento padrão para erros fatais é mostrar uma página para o erro interno do servidor (debug desativado) ou uma página com a mensagem, arquivo e linha (debug ativada).

Nota: Se você usar um manipulador de erros personalizado, as opções suportadas dependerão do seu manipulador.

`class ExceptionRenderer(`*Exception $exception*`)`

Alterando o tratamento de exceções

O tratamento de exceções oferece várias maneiras de personalizar como as exceções são tratadas. Cada abordagem fornece diferentes quantidades de controle sobre o processo de tratamento de exceções.

1. *Customize o template de error* Isso permite alterar os modelos de exibição renderizados como faria com qualquer outro modelo em seu aplicativo.
2. *Customize o ErrorController* Isso permite que você controle como as páginas de exceção são renderizadas.
3. *Customize o ExceptionRenderer* Isso permite que você controle como as páginas de exceção e o log são executados.
4. *Crie e registre seu próprio manipulador de erros* Isso fornece controle total sobre como os erros e exceções são tratados, registrados e renderizados.

Customizando Templates de Erro

O manipulador de erros padrão renderiza todas as exceções não capturadas que seu aplicativo gera com a ajuda de `Cake\Error\ExceptionRenderer` e o `ErrorController` do seu aplicativo.

As visualizações da página de erro estão localizadas em **templates/Error/**. Por padrão, todos os erros 4xx usam o modelo **error400.php** e todos os erros 5xx usam o **error500.php**. Seus modelos de erro terão as seguintes variáveis disponíveis:

- `message` A mensagem da exceção.
- `code` O código da exceção.
- `url` A URL requisitada.
- `error` O objeto da exceção.

No modo de depuração, se o erro estender `Cake\Core\Exception\Exception`, os dados retornados por `getAttributes()` serão expostos como variáveis de exibição também.

Nota: Você precisará definir debug para false, para ver seus modelos **error404** e **error500**. No modo de depuração, você verá a página de erro de desenvolvimento do CakePHP.

Personalizar o layout da página de erro

By default error templates use **templates/layout/error.php** for a layout. You can use the `layout` property to pick a different layout:

```
// inside templates/Error/error400.php
$this->layout = 'my_error';
```

The above would use **templates/layout/my_error.php** as the layout for your error pages.

Many exceptions raised by CakePHP will render specific view templates in debug mode. With debug turned off all exceptions raised by CakePHP will use either **error400.php** or **error500.php** based on their status code.

Customize the ErrorController

The `App\Controller\ErrorController` class is used by CakePHP's exception rendering to render the error page view and receives all the standard request life-cycle events. By modifying this class you can control which components are used and which templates are rendered.

If your application uses *routing-prefixes* you can create custom error controllers for each routing prefix. For example, if you had an `admin` prefix. You could create the following class:

```
namespace App\Controller\Admin;

use App\Controller\AppController;

class ErrorController extends AppController
{
    /**
     * Initialization hook method.
     *
     * @return void
     */
    public function initialize()
    {
        $this->loadComponent('RequestHandler');
    }

    /**
     * beforeRender callback.
     *
     * @param \Cake\Event\Event $event Event.
     * @return void
     */
    public function beforeRender(Event $event)
    {
        $this->viewBuilder()->setTemplatePath('Error');
    }
}
```

This controller would only be used when an error is encountered in a prefixed controller, and allows you to define prefix specific logic/templates as needed.

Change the ExceptionRenderer

If you want to control the entire exception rendering and logging process you can use the `Error.exceptionRenderer` option in **config/app.php** to choose a class that will render exception pages. Changing the ExceptionRenderer is useful when you want to provide custom error pages for application specific exception classes.

Your custom exception renderer class should be placed in **src/Error**. Let's assume our application uses `App\Exception\MissingWidgetException` to indicate a missing widget. We could create an exception renderer that renders specific error pages when this error is handled:

```
// In src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function missingWidget($error)
    {
        $response = $this->controller->response;

        return $response->withStringBody('Oops that widget is missing.');
    }
}

// In config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

The above would handle our `MissingWidgetException`, and allow us to provide custom display/handling logic for those application exceptions.

Exception rendering methods receive the handled exception as an argument, and should return a `Response` object. You can also implement methods to add additional logic when handling CakePHP errors:

```
// In src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    public function notFound($error)
    {
        // Do something with NotFoundException objects.
    }
}
```

Changing the ErrorController Class

The exception renderer dictates which controller is used for exception rendering. If you want to change which controller is used to render exceptions, override the `_getController()` method in your exception renderer:

```
// in src/Error/AppExceptionRenderer
namespace App\Error;

use App\Controller\SuperCustomErrorController;
use Cake\Error\ExceptionRenderer;

class AppExceptionRenderer extends ExceptionRenderer
{
    protected function _getController()
    {
        return new SuperCustomErrorController();
    }
}

// in config/app.php
'Error' => [
    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Creating your Own Error Handler

By replacing the error handler you can customize the entire error & exception handling process. By extending `Cake\Error\BaseErrorHandler` you can customize display logic more simply. As an example, we could build a class called `AppError` to handle our errors:

```
// In config/bootstrap.php
use App\Error\AppError;

$errorHandler = new AppError();
$errorHandler->register();

// In src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    public function _displayError($error, $debug)
    {
        echo 'There has been an error!';
    }

    public function _displayException($exception)
```

(continues on next page)

(continuação da página anterior)

```
    {
        echo 'There has been an exception!';
    }
}
```

The `BaseErrorHandler` defines two abstract methods. `_displayError()` is used when errors are triggered. The `_displayException()` method is called when there is an uncaught exception.

Changing Fatal Error Behavior

Error handlers convert fatal errors into exceptions and re-use the exception handling logic to render an error page. If you do not want to show the standard error page, you can override it:

```
// In src/Error/AppError.php
namespace App\Error;

use Cake\Error\BaseErrorHandler;

class AppError extends BaseErrorHandler
{
    // Other methods.

    public function handleFatalError($code, $description, $file, $line)
    {
        echo 'A fatal error has happened';
    }
}
```

Creating your own Application Exceptions

You can create your own application exceptions using any of the built in SPL exceptions[107], `Exception` itself, or *`Cake\Core\Exception\Exception`*. If your application contained the following exception:

```
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
}
```

You could provide nice development errors, by creating **templates/Error/missing_widget.php**. When in production mode, the above error would be treated as a 500 error and use the **error500** template.

If your exceptions have a code between `400` and `506` the exception code will be used as the HTTP response code.

The constructor for *`Cake\Core\Exception\Exception`* allows you to pass in additional data. This additional data is interpolated into the the `_messageTemplate`. This allows you to create data rich exceptions, that provide more context around your errors:

[107] https://php.net/manual/en/spl.exceptions.php

```php
use Cake\Core\Exception\Exception;

class MissingWidgetException extends Exception
{
    // Context data is interpolated into this format string.
    protected $_messageTemplate = 'Seems that %s is missing.';

    // You can set a default exception code as well.
    protected $_defaultCode = 404;
}

throw new MissingWidgetException(['widget' => 'Pointy']);
```

When rendered, this your view template would have a `$widget` variable set. If you cast the exception as a string or use its `getMessage()` method you will get `Seems that Pointy is missing.`.

Logging Exceptions

Using the built-in exception handling, you can log all the exceptions that are dealt with by ErrorHandler by setting the `log` option to `true` in your **config/app.php**. Enabling this will log every exception to `Cake\Log\Log` and the configured loggers.

Nota: If you are using a custom exception handler this setting will have no effect. Unless you reference it inside your implementation.

Built in Exceptions for CakePHP

HTTP Exceptions

There are several built-in exceptions inside CakePHP, outside of the internal framework exceptions, there are several exceptions for HTTP methods

exception `Cake\Http\Exception\`**BadRequestException**

Used for doing 400 Bad Request error.

exception `Cake\Http\Exception\`**UnauthorizedException**

Used for doing a 401 Unauthorized error.

exception `Cake\Http\Exception\`**ForbiddenException**

Used for doing a 403 Forbidden error.

exception `Cake\Http\Exception\`**InvalidCsrfTokenException**

Used for doing a 403 error caused by an invalid CSRF token.

exception `Cake\Http\Exception\`**NotFoundException**

Used for doing a 404 Not found error.

exception `Cake\Http\Exception\`**MethodNotAllowedException**

Used for doing a 405 Method Not Allowed error.

exception Cake\Http\Exception**NotAcceptableException**

Used for doing a 406 Not Acceptable error.

exception Cake\Http\Exception**ConflictException**

Used for doing a 409 Conflict error.

exception Cake\Http\Exception**GoneException**

Used for doing a 410 Gone error.

For more details on HTTP 4xx error status codes see **RFC 2616#section-10.4**[108].

exception Cake\Http\Exception**InternalErrorException**

Used for doing a 500 Internal Server Error.

exception Cake\Http\Exception**NotImplementedException**

Used for doing a 501 Not Implemented Errors.

exception Cake\Http\Exception**ServiceUnavailableException**

Used for doing a 503 Service Unavailable error.

For more details on HTTP 5xx error status codes see **RFC 2616#section-10.5**[109].

You can throw these exceptions from your controllers to indicate failure states, or HTTP errors. An example use of the HTTP exceptions could be rendering 404 pages for items that have not been found:

```
// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article not found'));
    }
    $this->set('article', $article);
    $this->set('_serialize', ['article']);
}
```

By using exceptions for HTTP errors, you can keep your code both clean, and give RESTful responses to client applications and users.

Using HTTP Exceptions in your Controllers

You can throw any of the HTTP related exceptions from your controller actions to indicate failure states. For example:

```
use Cake\Network\Exception\NotFoundException;

public function view($id = null)
{
    $article = $this->Articles->findById($id)->first();
    if (empty($article)) {
        throw new NotFoundException(__('Article not found'));
    }
```

(continues on next page)

[108] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.4
[109] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.5

(continuação da página anterior)

```
    $this->set('article', 'article');
    $this->set('_serialize', ['article']);
}
```

The above would cause the configured exception handler to catch and process the *NotFoundException*. By default this will create an error page, and log the exception.

Other Built In Exceptions

In addition, CakePHP uses the following exceptions:

exception Cake\View\Exception**MissingViewException**
: The chosen view class could not be found.

exception Cake\View\Exception**MissingTemplateException**
: The chosen template file could not be found.

exception Cake\View\Exception**MissingLayoutException**
: The chosen layout could not be found.

exception Cake\View\Exception**MissingHelperException**
: The chosen helper could not be found.

exception Cake\View\Exception**MissingElementException**
: The chosen element file could not be found.

exception Cake\View\Exception**MissingCellException**
: The chosen cell class could not be found.

exception Cake\View\Exception**MissingCellViewException**
: The chosen cell view file could not be found.

exception Cake\Controller\Exception**MissingComponentException**
: A configured component could not be found.

exception Cake\Controller\Exception**MissingActionException**
: The requested controller action could not be found.

exception Cake\Controller\Exception**PrivateActionException**
: Accessing private/protected/_ prefixed actions.

exception Cake\Console\Exception**ConsoleException**
: A console library class encounter an error.

exception Cake\Console\Exception**MissingTaskException**
: A configured task could not found.

exception Cake\Console\Exception**MissingShellException**
: The shell class could not be found.

exception Cake\Console\Exception**MissingShellMethodException**
: The chosen shell class has no method of that name.

exception Cake\Database\Exception**MissingConnectionException**
: A model's connection is missing.

exception Cake\Database\Exception**MissingDriverException**

A database driver could not be found.

exception Cake\Database\Exception**MissingExtensionException**

A PHP extension is missing for the database driver.

exception Cake\ORM\Exception**MissingTableException**

A model's table could not be found.

exception Cake\ORM\Exception**MissingEntityException**

A model's entity could not be found.

exception Cake\ORM\Exception**MissingBehaviorException**

A model's behavior could not be found.

exception Cake\ORM\Exception**PersistenceFailedException**

An entity couldn't be saved/deleted while using *Cake\ORM\Table::saveOrFail()* or *Cake\ORM\Table::deleteOrFail()*.

exception Cake\Datasource\Exception**RecordNotFoundException**

The requested record could not be found. This will also set HTTP response headers to 404.

exception Cake\Routing\Exception**MissingControllerException**

The requested controller could not be found.

exception Cake\Routing\Exception**MissingRouteException**

The requested URL cannot be reverse routed or cannot be parsed.

exception Cake\Routing\Exception**MissingDispatcherFilterException**

The dispatcher filter could not be found.

exception Cake\Core\Exception**Exception**

Base exception class in CakePHP. All framework layer exceptions thrown by CakePHP will extend this class.

These exception classes all extend *Exception*. By extending Exception, you can create your own "framework" errors.

Cake\Core\Exception\Exception::**responseHeader**(*$header = null*, *$value = null*)

See Cake\Network\Request::header()

All Http and Cake exceptions extend the Exception class, which has a method to add headers to the response. For instance when throwing a 405 MethodNotAllowedException the rfc2616 says:

```
"The response MUST include an Allow header containing a list of valid
methods for the requested resource."
```

CAPÍTULO 20

Sistema de Eventos

Criar aplicações com facilidade de manutenção é uma ciência e uma arte ao mesmo tempo. É de conhecimento geral que a chave para ter um código de qualidade é fazer objetos desacoplados e coesos ao mesmo tempo. Coesão significa que todos os metodos e propriedades de uma classe são fortemente relacionados entre classes em sí e não estão tentando fazer o trabalho que deveria ser feito por outros objetos, equanto o desacoplamento é a medida de quão «estranha» uma classe é para objetos externos e o quanto essa classe depende desses objetos.

Existem alguns casos onde você precisa se comunicar com outras partes da aplicação, sem existir dependências diretamente no código («hardcoded»), diminuindo, assim, a coesão e aumentando o acoplamento. Usar o padrão Observer, que permite que objetos sejam notificados por outros objetos e ouvintes anônimos sobre mudanças. Observer é um padrão que ajuda a atingir esse objetivo.

Ouvintes no padrão observer podem se inscrever para eventos e escolher se deve agir, caso seja relevante. Se você já usou JavaScript tem uma boa chance de que você já esteja familiarizado com programação orientada a eventos.

O CakePHP emula vários desses aspectos de quando objetos são engatilhados e gerenciados em bibliotecas populares de JavaScript, como jQuery. Na implementação do CakePHP um evento é disparado para todos os listeners (ouvintes). O objeto event tem as informações do evento e a habilidade de parar a propagação de um evento em qualquer ponto do evento. Ouvintes podem se registrar ou delegar essa tarefa para outros objetos e tem a chance de alterar o estado do evento em si pelo resto dos callbacks.

O subsistema de eventos é o coração dos callbacks de Model, Behavior, Controller, View e Helper. Se você já usou um deles, você já está de alguma forma familiarizado com os eventos no CakePHP.

Exemplo de Uso dos Eventos

Vamos assumir que você está construindo um plugin de carrinho de compras e gostaria de focar somente na lógica de lidar com o pedido. Você não quer incluir nenhuma lógica de envios, notificação dos usuários ou incrementar/remover um item do estoque. Mas, essas são tarefas importantes para pessoas que vão usar o seu plugin. Se você não estivesse usando eventos, você poderia tentar implementar isso incluindo Bahaviors no seu Model, ou adicionando Components no seu Controller. Fazer isso é um desvio na maioria das vezes, já que você teria que adicionar código para carregar externamente esses Behaviors, ou adicionar hooks ao Controller do seu plugin.

Você pode usar eventos para permitir que você separe as responsabilidades do seu código e permitir que outras responsabilidades se inscrevam nos eventos do seu plugin. Por exemplo, no plugin de carrinho você tem um model Orders que cria os pedidos, você gostaria de notificar o resto da aplicação que um pedido foi criado, para manter o Model Orders limpo você poderia usar eventos:

```
// Cart/Model/Table/OrdersTable.php
namespace Cart\Model\Table;

use Cake\Event\Event;
use Cake\ORM\Table;

class OrdersTable extends Table
{
    public function place($order)
    {
        if ($this->save($order)) {
            $this->Cart->remove($order);
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->eventManager()->dispatch($event);

            return true;
        }

        return false;
    }
}
```

Obsoleto desde a versão 3.5.0: Use `getEventManager()`.

O exemplo acima permite você notificar outras partes da aplicação em que um pedido foi feito e você pode então, enviar emails, notificações, atualizar o estoque, fazer o log das estatísticas relevantes e outras tarefas em um objeto separado que foca nessas responsabilidades.

Acessando os Gerenciadores de Evento (Event Menagers)

No CakePHP os eventos são disparados para os gerenciadores de evento (event managers). Gerenciadores de evento disponíveis estão em todas as Table, View e Controller, utilizando `getEventManager()`:

```
$events = $this->getEventManager();
```

Cada Model tem o seu próprio gerenciador de evento, enquando View e Controller compartilham o mesmo, Isso permite que os eventos dos Models sejam isolados, e permitem os Components ou Controller reagirem a eventos criados na View, caso necessário.

Gerenciador de Eventos Global

Adicionado aos gerenciadores de evento no nível da instância, o CakePHP provê um gerenciador de evento global, que permite ouvir a qualquer evento disparado pela aplicação. isso é útil quando anexar Ouvintes a uma instancia pode ser incômodo ou difícil. O gerenciador de eventos global é um singleton de `Cake\Event\EventManager`. Ouvintes anexados ao gerenciador de eventos global são executados antes dos Ouvintes de instâncias com a mesma prioridade. você pode acessar o gerenciador de eventos glocal utilizando o metodo estático:

```
// Em qualquer arquivo de configuração ou arquivo que seja executado *antes* do evento
use Cake\Event\EventManager;

EventManager::instance()->on(
    'Model.Order.afterPlace',
    $aCallback
);
```

Uma coisa que deve ser levada em conta é que existem eventos com o mesmo nome, mas com assuntos divergentes, então verificar se o evento é requerido em qualquer função que é anexada globalmente, desse modo, evitando bugs, lembre-se que com a flexibilidade de um gerenciador de evento global, uma certa complexidade é adicionada.

O metodo `Cake\Event\EventManager::dispatch()` aceita o objeto do evento como um argumento, e notifica a todos os Ouvintes e Callbacks parando esse objeto adiante. Os Ouvintes vão lidar com toda a lógica extra ligada ao evento `afterPlace`, você pode, enviar emails, atualizar estatísticas do usuário em objetos separados, ou também delegar isso para tarefas offline que você possa precisar.

Rastreando Eventos

Para manter uma lista de eventos que são disparados em um `EventManager`, você pode habilitar o rastreamento de eventos (event tracking). Para fazer isso anexe um `Cake\Event\EventList` ao gerenciador:

```
EventManager::instance()->setEventList(new EventList());
```

Após disparar um evento para o gerenciador você pode recuperar ele da lista de eventos:

```
$eventsFired = EventManager::instance()->getEventList();
$firstEvent = $eventsFired[0];
```

O rastreamento de eventos pode ser desabilitado ao remover a lista de eventos ou chamando `Cake\Event\EventList::trackEvents(false)`.

Eventos do Core

Existem vários eventos que fazem parte do core do framework o qual a sua aplicação pode ouvir. Cada camada do CakePHP emite um evento que você pode utilizar na sua aplicação.

- *ORM/Model events*
- *Controller events*
- *View events*

Registrando Listeners

Listeners são o meio preferido para registrar callbacks de qualquer evento. Isso é feito implementando a interface `Cake\Event\EventListenerInterface` em qualquer classe que você deseje registrar um callback. Classes implementando a interface devem ter o metodo `implementedEvents()`. Esse método deve retornar um array associativo com o nome de todos os eventos que a classe vai gerenciar.

Para continuar o exemplo anterior, vamos imaginas que temos uma classe UserStatistic responsável por calcular o histórico de compras do usuário, e compilar nas estatísticas globais do site. Esse é um ótimo exemplo de onde usar uma classe Listener. Fazendo isso permite você se concentrar nas lógica das estatísticas em um local e responder ao eventos como necessários. Nosso listener `UserStatistics` pode começar como abaixo:

```
use Cake\Event\EventListenerInterface;

class UserStatistic implements EventListenerInterface
{
    public function implementedEvents()
    {
        return [
            'Model.Order.afterPlace' => 'updateBuyStatistic',
        ];
    }

    public function updateBuyStatistic($event, $order)
    {
        // Código para atualizar as estatísticas

        // Code to update statistics
    }
}

// Anexa o objeto UserStatistic para o gerenciador de evento da Order
$statistics = new UserStatistic();
$this->Orders->getEventManager()->on($statistics);
```

Como você pôde ver nó código acima, o metodo `on()` aceita instancias da interface `EventListener`. Internamente o gerenciador de eventos vai utilizar os `implementedEvents()` para anexar ao callback corretamente.

Registrando Listeners Anônimos

Enquanto objeto de Event Listerners são geralmente um melhor método para implementar Listeners você pode utilizar uma `callable` como Event Listener. Por exemplo, se nós quisessemos colocar qualquer pedido nos arquivos de log, nós poderiamos utilizar uma função anônima para isso:

```
use Cake\Log\Log;

$this->Orders->getEventManager()->on('Model.Order.afterPlace', function ($event) {
    Log::write(
        'info',
        'A new order was placed with id: ' . $event->getSubject()->id
    );
});
```

Além de funções anônimas você pode usar qualquer outro callable no qual o PHP suporta:

```
$events = [
    'email-sending' => 'EmailSender::sendBuyEmail',
    'inventory' => [$this->InventoryManager, 'decrement'],
];
foreach ($events as $callable) {
    $eventManager->on('Model.Order.afterPlace', $callable);
}
```

Quando trabalhamos com plugins que não dispara eventos especificos, você pode utilizar Event Listeners dos eventos padrão. Vamos pensar, por exemplo o plugin "UserFeedback" que lida com o feedback dos usuários. A partir da sua aplicação, você poderia querer saber quando um feedback foi salvo no banco de dados e intervir nele. Você pode utilizar o gerenciador de eventos global para pegar o evento `Model.afterSave`. No entendo, você pode pegar um caminho mais direto. e escutar somente o que você realmente precisa:

```
// Você pode criar o código a seguir antes de persistir os dados no banco
// exemplo no config/bootstrap.php

use Cake\ORM\TableRegistry;
// Se está enviando emails
use Cake\Mailer\Email;

TableRegistry::getTableLocator()->get('ThirdPartyPlugin.Feedbacks')
    ->getEventManager()
    ->on('Model.afterSave', function($event, $entity)
    {
            // Por exemplo, podemos mandar um email para o admin
            // Antes da versão 3.4 use os métodos from()/to()/subject()
        $email = new Email('default');
        $email->setFrom(['info@yoursite.com' => 'Your Site'])
            ->setTo('admin@yoursite.com')
            ->setSubject('New Feedback - Your Site')
            ->send('Body of message');
    });
```

Você pode usar esse mesmo método para ligar a objetos Listener.

Interagindo com Listeners Existentes

Supondo que vários ouvintes de eventos tenham sido registrados, a presença ou ausência de um padrão de evento específico pode ser usada como base de alguma ação:

```
// Anexa Listeners ao EventManager.
$this->getEventManager()->on('User.Registration', [$this, 'userRegistration']);
$this->getEventManager()->on('User.Verification', [$this, 'userVerification']);
$this->getEventManager()->on('User.Authorization', [$this, 'userAuthorization']);

// Em algum outro local da sua aplicação.
$events = $this->getEventManager()->matchingListeners('Verification');
if (!empty($events)) {
    // Executa a lógica relacionada a precença do Event Listener 'Verification'.
    // Por exemplo, remover o Listener caso esteja presente.
    $this->getEventManager()->off('User.Verification');
} else {
    // Executa a lógica relacionada a ausencia do event listener 'Verification'
}
```

Nota: O padrão passado para o método `matchingListeners` é case sensitive.

Estabelecendo Prioridades

Em alguns casos você pode querer controlar a ordem em que os Listeners são invocados, por exemplo, se nós voltarmos ao nosso exemplo das estatísticas do usuários. Seria ideal se esse Listener fosse chamado no final da pilha. Ao chamar no final do pilha de ouvintes, nós garantimos que o evento não foi cancelado e que, nenhum outro listeners retornou exceptions. Nós podemos também pegar o estado final dos objetos, no caso de outros ouvintes possam terem modificado o objeto de assunto ou do evento.

Prioridades são definidas como inteiros (integer) quando adicionadas ao ouvinte. Quando maior for o número, mais tarde esse metodo será disparado. A prioridade padrão para todos os listeners é `10`. Se você precisa que o seu método seja executado antes, utilize um valor menor que o padrão. Por outro lado se você deseja rodar o seu callback depois dos outros, usando um número acima de `10` será suficiente.

Se dois callbacks tiverem a mesma prioridade, eles serão executados de acordo com a ordem em que foram adicionados. Você pode definir as prioridades utilizando o método `on()` para callbacks, e declarando no método `implementedEvents()` para os Event Listeners:

```
// Definindo a prioridade para um callback
$callback = [$this, 'doSomething'];
$this->getEventManager()->on(
    'Model.Order.afterPlace',
    ['priority' => 2],
    $callback
);

// Definindo a prioridade para um Listener
class UserStatistic implements EventListenerInterface
{
    public function implementedEvents()
    {
```

(continues on next page)

(continuação da página anterior)

```
        return [
            'Model.Order.afterPlace' => [
                'callable' => 'updateBuyStatistic',
                'priority' => 100
            ],
        ];
    }
}
```

Como você pôde ver, a principal diferença entre objetos `EventListener` é que você precisa usar uma array para especificar o metodo callable e a preferência de prioridade. A chave `callable` é uma array especial que o gerenciador vai ler para saber qual função na classe ele deverá chamar.

Obtendo Dados do Evento como Argumentos da Função

Quando eventos tem dados definidos no seu construtor, esses dados são convertidos em argumentos para os ouvintes. Um exemplo da camada ViewView é o afterRender callback:

```
$this->getEventManager()
    ->dispatch(new Event('View.afterRender', $this, ['view' => $viewFileName]));
```

Os ouvintes do callback `View.afterRender` devem ter a seguinte assinatura:

```
function (Event $event, $viewFileName)
```

Cada valor fornecido ao construtor Event será convertido em parâmetros de função na ordem em que aparecem na matriz de dados. Se você usar uma matriz associativa, o resultado `array_values` determinará a ordem dos argumentos da função.

Nota: Diferente do CakePHP 2.x, converter dados para os arqumentos do listener é o comportamento padrão e não pode ser desativado.

Disparando Eventos

Uma vez que você tem uma instancia do event manager você pode disparar eventos utilizando `dispatch()`. Esse método aceita uma instancia da class `Cake\Event\Event`. Vamos ver como disparar um evento:

```
// Um event listener tem que ser instanciado antes de disparar um evento.
// Crie um evento e dispare ele.
$event = new Event('Model.Order.afterPlace', $this, [
    'order' => $order
]);
$this->getEventManager()->dispatch($event);
```

`Cake\Event\Event` aceita três argumentos no seu construtor. O primeiro é o nome do evento, você deve tentar manter esse nome o mais único possível, ainda assim, deve ser de fácil entendimento . Nós sugerimos a seguinte convenção: `Camada.nomeDoEvento` para eventos acontecendo a nível de uma camada (ex. `Controller.startup`, `View.beforeRender`) e `Camada.Classe.NomeDoEvento` para eventos que acontecen em uma classe especifica em uma camada, exemplo `Model.User.afterRegister` ou `Controller.Courses.invalidAccess`.

O segundo argumento é o `subject`, ou seja, o objeto associado ao evento, geralmente quando é a mesma classe que desencadeia eventos sobre si mesmo, o uso de `$this` será o caso mais comum. Embora um componente também possa disparar eventos do controlador. A classe de assunto é importante porque os ouvintes terão acesso imediato às propriedades do objeto e terão a chance de inspecioná-las ou alterá-las rapidamente.

Finalmente o terceiro argumento é qualquer dado adicional que você deseja enviar ao evento. Esses dados podem ser qualquer coisa que você considere útil enviar aos listeners. Enquanto esse argumento pode ser de qualquer tipo, nós recomendamos que seja uma array associativa.

O medoto `dispatch()` aceita um objeto de evento como argumento e notifica a todos os listeners inscritos.

Parando Eventos

Assim como nos eventos do DOM, você pode querer parar um evento para previnir que outros listeners sejam notificados. Você pode ver isso em ação nos Callbacks do model (ex. beforeSave) onde é possível parar o operação de persistir os dados se o código decidir que não pode continuar

Para parar um evento você pode retornar `false` nos seus callbacks ou chamar o método `stopPropagation()` no objeto do evento:

```
public function doSomething($event)
{
    // ...
    return false; // Para o evento
}

public function updateBuyStatistic($event)
{
    // ...
    $event->stopPropagation();
}
```

Parar um evento vai previnir que qualquer callback adicional seja chamado. Além disso o código que disparou o evento pode se comportar de maneira diferente baseado no evento sendo parado ou não. Geralmente não faz sentido parar "depois" do evento, mas parar "antes" do evento costuma ser usado para impedir toda a operação de acontecer.

Para verificar se um evento foi parado você pode chamar o metodo `isStopped()` no objeto do evento object:

```
public function place($order)
{
    $event = new Event('Model.Order.beforePlace', $this, ['order' => $order]);
    $this->getEventManager()->dispatch($event);
    if ($event->isStopped()) {
        return false;
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

No exemplo anterior o pedido não será salvo se o evento for parado durante o processamento do callback `beforePlace`.

Parando o Resultado de um Evento

Toda vez que um callback retorna um valor não nulo ou não falso, ele é armazenado na propriedade `$result` do objeto do evento. Isso é útil quando você quer permitir callbacks a modificar a execução do evento. Vajamos novamente nosso exemplo `beforePlace``e vamos deixar os callbacks modififcar os dados de ``$order`.

Resultados de eventos podem ser alterados utilizando o resultado do objeto do evento diretamente ou retornando o valor no próprio callback:

```
// Um callback de ouvinte
public function doSomething($event)
{
    // ...
    $alteredData = $event->getData('order') + $moreData;

    return $alteredData;
}
// Outro callback
public function doSomethingElse($event)
{
    // ...
    $event->setResult(['order' => $alteredData] + $this->result());
}

// Utilizando o resultado do evento
public function place($order)
{
    $event = new Event('Model.Order.beforePlace', $this, ['order' => $order]);
    $this->getEventManager()->dispatch($event);
    if (!empty($event->getResult()['order'])) {
        $order = $event->getResult()['order'];
    }
    if ($this->Orders->save($order)) {
        // ...
    }
    // ...
}
```

É possível alterar qualquer propriedade do objeto de evento e passar os novos dados para o próximo retorno de chamada. Na maioria dos casos, fornecer objetos como dados ou resultado de eventos e alterar diretamente o objeto é a melhor solução, pois a referência é mantida a mesma e as modificações são compartilhadas em todas as chamadas de retorno de chamada.

Removento Callbacks e Ouvintes

Se por qualquer motivo você desejar remover os callbacks do gerenciador de eventos é só chamar o método `Cake\Event\EventManager::off()` utilizando como argumentos os dois primeiros parâmetros usados para anexá-lo:

```
// Adicionando uma função
$this->getEventManager()->on('My.event', [$this, 'doSomething']);

// Removendo uma função
$this->getEventManager()->off('My.event', [$this, 'doSomething']);
```

(continues on next page)

(continuação da página anterior)

```
// Adicionando uma função anônima.
$myFunction = function ($event) { ... };
$this->getEventManager()->on('My.event', $myFunction);

// Removendo uma função anônima
$this->getEventManager()->off('My.event', $myFunction);

// Adicionando um EventListener
$listener = new MyEventLister();
$this->getEventManager()->on($listener);

// Removendo uma única chave de um evento em um ouvinte
$this->getEventManager()->off('My.event', $listener);

// Removento todos os callbacks implemantados por um ouvinte
$this->getEventManager()->off($listener);
```

Eventos são uma ótima maneira de separar responsabilidades na sua aplicação e fazer com que classes sejam coesas e desacopladas. Eventos podem ser utilizados para desacoplar o código de uma aplicação e fazer extensão via plugins.

Lembre-se de que com grande poder vem uma grande responsabilidade. Usar muitos eventos pode dificultar a depuração e exigir testes adicionais de integração.

Leitura Adicional

- *Behaviors (Comportamentos)*
- *Componentes*
- *Helpers (Facilitadores)*
- *Testando Eventos*

CAPÍTULO 21

Internacionalização e Localização

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[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.

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 (internacionalization) 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:

[110] https://github.com/cakephp/docs

```
<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.

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[111]. Os arquivos precisam ser colocado dentro do Diretório **resources/locales/** e dentro deste diretório, deve haver uma subpasta para cada idioma, por exemplo:

```
/resources
    /locales
        /en_US
            default.po
        /en_GB
            default.po
            validation.po
        /es
            default.po
```

O domínio padrão é "default", portanto, a pasta **resources/locales/** 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 **resources/locales/**. O CakePHP localized library[112] 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: **resources/locales/<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
    /resources
        /locales
            /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}"
```

(continues on next page)

[111] https://en.wikipedia.org/wiki/Gettext
[112] https://github.com/cakephp/localized

(continuação da página anterior)

```
msgid "I'm {0,number} years old"
msgstr "J'ai {0,number} ans"
```

Nota: As traduções são armazenadas em cache - Certifique-se de que você sempre limpa o cache após fazer alterações nas traduções! Você pode utilizar a ferramenta *cache tool* e executar por exemplo `bin/cake cache clear _cake_core_`, ou limpar manualmente a pasta `tmp/cache/persistent` (se estiver utilizando o cache baseado em arquivo).

Extraindo arquivos .pot com I18n Shell

Para criar os arquivos .pot apartir 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.

Definir a localidade padrão

A localidade padrão pode ser definida em no arquivo **config/app.php**, definindo `App.default Locale`:

```
'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::setLocale('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 caracter ' (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[113] 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
```

[113] https://php.net/manual/en/messageformatter.format.php

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

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 seqüência:

```
__('{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 seqüência 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[114] 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);

// 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[115] para uma explicação detalhada dos números de formulário de plurais para cada idioma.

[114] https://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html
[115] https://translations.launchpad.net/+languages

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::setLocale('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:

```
use Cake\I18n\MessagesFileLoader as Loader;

// Load messages from resources/locales/folder/sub_folder/filename.po

I18n::translator(
    'animals',
    'fr_FR',
    new Loader('filename', 'folder/sub_folder', 'po')
);
```

CAPÍTULO 22

Logging

Logging Configuration

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[116] 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.

[116] https://github.com/cakephp/docs

CAPÍTULO 23

Formulários sem Models

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.

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']);
    }
```

(continues on next page)

(continuação da página anterior)

```
    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 você 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.');
            }
```

(continues on next page)

(continuação da página anterior)

```
        }
        $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 requesição é do tipo GET, caso contrário você sobreescreverá 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.

CAPÍTULO 24

Plugins

O CakePHP permite que você configure uma combinação de controllers, models e views, que são os plugins 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 torná-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 <https://packagist.org>* _ E podem ser instalados com o `Composer`. Para instalar o DebugKit, você deve fazer assim o assim:

```
php composer.phar require cakephp/debug_kit
```

Ou se o composer tiver sido instalado globalmente assim:

composer require cakephp/debug_kit

Isso instalaria a versão mais recente do DebugKit e atualizaria seus arquivos **composer.json**, **composer.lock**, atualiza **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**, no raiz do aplicativo. Supondo que você deseja instalar um plugin chamado "ContactManager", você deve ter uma pasta em **plugins** chamado "ContactManager". Neste diretório existe o src, o 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 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, com Composer e `plugin-installer` O pacote o gerenciará 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]);
```

(continues on next page)

(continuação da página anterior)

```
    // 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 /bake/usage 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`.

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 qrquivos 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
{
```

(continues on next page)

(continuação da página anterior)

```
    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;

// Prior to 3.6 use TableRegistry::get('ContactManager.Contacts')
$contacts = TableRegistry::getTableLocator()->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]/templates/`. Para nós o plugin ContactManager, precisamos de uma view para o nosso `ContactsController::index()` action, então incluamos isso também:

```
// plugins/ContactManager/templates/Contacts/index.php:
<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]/templates/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 **templates/plugin/[Plugin]/[Controller]/[view].php**. Para o controller Contacts você pode fazer o seguinte arquivo:

```
templates/plugin/ContactManager/Contacts/index.php
```

Criar este arquivo permitiria que você substituir **plugins/ContactManager/templates/Contacts/index.php**.

Se o seu plugin estiver em uma dependência no composer (ou seja, "TheVendor/ThePlugin"), o caminho para da view "index" do controller personalizado será:

```
templates/plugin/TheVendor/ThePlugin/Custom/index.php
```

Criar este arquivo permitiria que você substituir **vendor/thevendor/theplugin/templates/Custom/index.php**.

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:

```
templates/plugin/ContactManager/Admin/ContactManager/index.php
```

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 *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');
```

(continues on next page)

(continuação da página anterior)

```
// 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 aplicacao 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[117]. Desta forma, outras pessoas podem Use-o como dependência do compositor. Você também pode propor seu plugin para o Lista de awesome-cakephp list[118].

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.

[117] https://plugins.cakephp.org
[118] https://github.com/FriendsOfCake/awesome-cakephp

CAPÍTULO 25

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');
```

(continues on next page)

(continuação da página anterior)

```
        $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)) {
```

(continues on next page)

(continuação da página anterior)

```
            $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 buld-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 saidas REST para nosso RecipesController dentro de **templates/Recipes/xml**. Nós também podemos usar The `Xml` para saída XML rápida e fácil:

```
// templates/Recipes/xml/index.php
// 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 yummmy</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.

CAPÍTULO 26

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[119] 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[120] ou mcrypt[121] dependendo de qual deles estiver disponível em seu sistema. Dados criptografados em uma implementação são portáveis para a outra.

Aviso: A extensão mcrypt[122] foi considerada obsoleta no PHP7.1

[119] https://api.cakephp.org/4.x/class-Cake.Utility.Security.html
[120] https://php.net/openssl
[121] https://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 *hash()*. Um exemplo de uso pode ser:

```
// Assumindo que $key está gravada em algum lugar, ela pode ser reusada para
// descriptografar depois.
$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
$sha1 = Security::hash('CakePHP Framework', 'sha1', true);

// Usando um salt customizado
$sha1 = Security::hash('CakePHP Framework', 'sha1', 'my-salt');
```

(continues on next page)

[122] https://php.net/mcrypt

(continuação da página anterior)

```
// 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*)

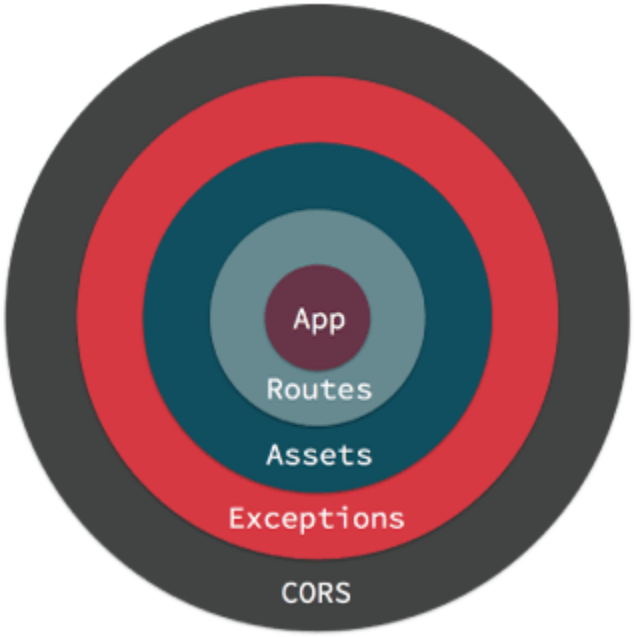
Obter `$length` número de bytes de uma fonte segura aleatória. Esta função utiliza um dos seguntes métodos:

- Função `random_bytes` do PHP.
- Função `openssl_random_pseudo_bytes` da extensão SSL.

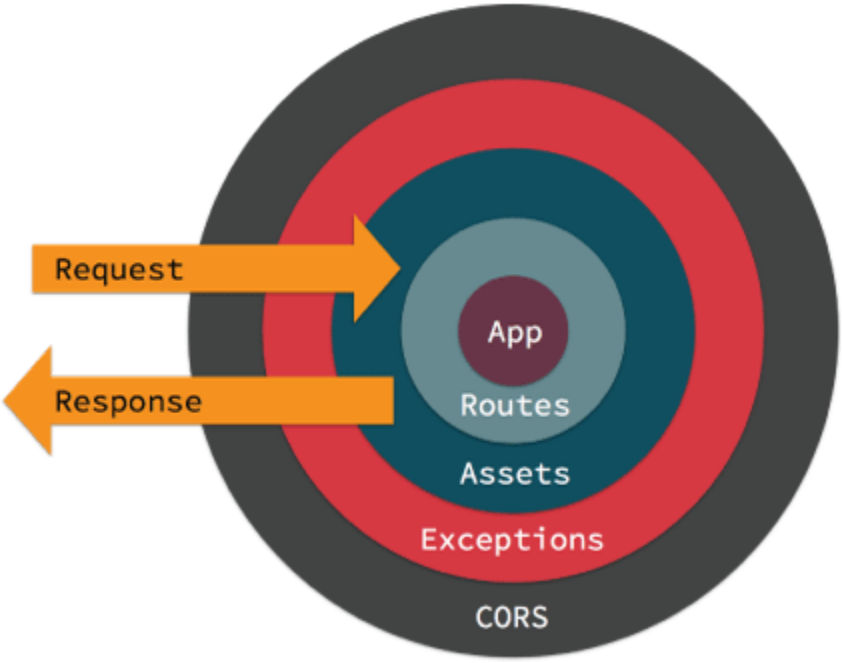
Se nenhuma das opções estiverem disponíveis um `warning` será emitido e um valor não seguro será usado por motivos de compatibilidade.

Middleware

Os objetos de middleware permitem que você "embrulhe" seu aplicativo em camadas reutilizáveis e composíveis de manipulação de solicitações ou lógica de criação de respostas. Visualmente, seu aplicativo termina no centro e o middleware é envolvido em volta do aplicativo como uma cebola. Aqui, podemos ver um aplicativo agrupado com os middlewares Routes, Assets, Exception Handling e CORS.



Quando um pedido é tratado pelo seu aplicativo, ele entra no middleware mais externo. Cada middleware pode delegar a solicitação/resposta para a próxima camada ou retornar uma resposta. O retorno de uma resposta impede que as camadas inferiores vejam a solicitação. Um exemplo disso é o plugin AssetMiddleware manipulando uma solicitação de uma imagem de durante o desenvolvimento.



Se nenhum middleware executar uma ação para manipular a solicitação, um controlador será localizado e terá sua ação invocada ou uma exceção será gerada gerando uma página de erro.

O middleware faz parte da nova pilha HTTP no CakePHP que aproveita as interfaces de solicitação e resposta PSR-7. O CakePHP também suporta o padrão PSR-15 para manipuladores de solicitações de servidor, para que você possa

usar qualquer middleware compatível com PSR-15 disponível em The Packagist[123].

Middleware em CakePHP

O CakePHP fornece vários middlewares para lidar com tarefas comuns em aplicativos da web:

- `Cake\Error\Middleware\ErrorHandlerMiddleware` intercepta exceções do middleware empacotado e renderiza uma página de erro usando o manipulador de exceção *Erros & Exceções*.
- `Cake\Routing\AssetMiddleware` verifica se a solicitação está se referindo a um tema ou arquivo estático do plug-in, como CSS, JavaScript ou arquivo de imagem armazenado na pasta raiz da web de um plug-in ou na pasta correspondente a um Tema.
- `Cake\Routing\Middleware\RoutingMiddleware` usa o `Router` para analisar a URL recebida e atribuir parâmetros de roteamento à solicitação.
- `Cake\I18n\Middleware\LocaleSelectorMiddleware` habilita a troca automática de idioma no cabeçalho `Accept-Language` enviado pelo navegador.
- `Cake\Http\Middleware\SecurityHeadersMiddleware` facilita adicionar cabeçalhos relacionados à segurança como `X-Frame-Options` às respostas.
- `Cake\Http\Middleware\EncryptedCookieMiddleware` oferece a capacidade de manipular cookies criptografados, caso você precise manipular cookies com dados ofuscados.
- `Cake\Http\Middleware\CsrfProtectionMiddleware` adiciona proteção CSRF ao seu aplicativo.
- `Cake\Http\Middleware\BodyParserMiddleware` permite decodificar JSON, XML e outros corpos de solicitação codificados com base no cabeçalho `Content-Type`.
- `Cake\Http\Middleware\CspMiddleware` simplifica a adição de cabeçalhos de política de segurança de conteúdo ao seu aplicativo.

Usando Middleware

O middleware pode ser aplicado ao seu aplicativo globalmente ou individualmente a escopos de roteamento.

Para aplicar o middleware a todas as solicitações, use o método `middleware` da sua classe `App\Application`. O método de gancho `middleware` do seu aplicativo será chamado no início do processo de solicitação; você pode usar o objeto `MiddlewareQueue` para anexar o middleware:

```
namespace App;

use Cake\Http\BaseApplication;
use Cake\Http\MiddlewareQueue;
use Cake\Error\Middleware\ErrorHandlerMiddleware;

class Application extends BaseApplication
{
    public function middleware(MiddlewareQueue $middlwareQueue): MiddlewareQueue
    {
        // Vincule o manipulador de erros à fila do middleware.
        $middlwareQueue->add(new ErrorHandlerMiddleware());

        return $middlwareQueue;
```

(continues on next page)

[123] https://packagist.org

(continuação da página anterior)

```
    }
}
```

Além de adicionar ao final do `MiddlewareQueue`, você pode executar várias operações:

```
$layer = new \App\Middleware\CustomMiddleware;

// O middleware adicionado será o último da fila.
$middlwareQueue->add($layer);

// O middleware precedido será o primeiro da fila.
$middlwareQueue->prepend($layer);

// Insira em um slot específico. Se o slot estiver
// fora dos limites, ele será adicionado ao final.
$middlwareQueue->insertAt(2, $layer);

// Insira antes de outro middleware.
// Se a classe nomeada não puder ser encontrada,
// uma exceção será gerada.
$middlwareQueue->insertBefore(
    'Cake\Error\Middleware\ErrorHandlerMiddleware',
    $layer
);

// Insira depois de outro middleware.
// Se a classe nomeada não puder ser encontrada, o
// o middleware será adicionado ao final.
$middlwareQueue->insertAfter(
    'Cake\Error\Middleware\ErrorHandlerMiddleware',
    $layer
);
```

Além de aplicar o middleware a todo o aplicativo, você pode aplicar o middleware a conjuntos específicos de rotas usando *Scope Middleware*.

Adicionando Middleware a partir de Plugins

Os plug-ins podem usar seu método de gancho `middleware` para aplicar qualquer middleware que eles tenham à fila de middleware do aplicativo:

```
// Em plugins/ContactManager/src/Plugin.php
namespace ContactManager;

use Cake\Core\BasePlugin;
use Cake\Http\MiddlewareQueue;
use ContactManager\Middleware\ContactManagerContextMiddleware;

class Plugin extends BasePlugin
{
    public function middleware(MiddlewareQueue $middlwareQueue): MiddlewareQueue
    {
```

(continues on next page)

(continuação da página anterior)

```
        $middlwareQueue->add(new ContactManagerContextMiddleware());

        return $middlwareQueue;
    }
}
```

Criando um Middleware

O middleware pode ser implementado como funções anônimas (Closures) ou classes que estendem `Psr\Http\Server\MiddlewareInterface`. Embora os Closures sejam adequados para tarefas menores, eles tornam os testes mais difíceis e podem criar uma classe `Application` complicada. As classes de middleware no CakePHP têm algumas convenções:

- Os arquivos de classe Middleware devem ser colocados em ** src/Middleware**. Por exemplo: **src/Middleware/CorsMiddleware.php**
- As classes de middleware devem ter o sufixo `Middleware`. Por exemplo: `LinkMiddleware`.
- O Middleware deve implementar `Psr\Http\Server\MiddlewareInterface`.

O middleware pode retornar uma resposta chamando `$handler->handle()` ou criando sua própria resposta. Podemos ver as duas opções em nosso middleware simples:

```
// Em src/Middleware/TrackingCookieMiddleware.php
namespace App\Middleware;

use Cake\Http\Cookie\Cookie;
use Cake\I18n\Time;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\RequestHandlerInterface;
use Psr\Http\Server\MiddlewareInterface;

class TrackingCookieMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface
    {
        // Chamar $handler->handle() delega o controle para
        // o *próximo* middleware na fila do seu aplicativo.
        $response = $handler->handle($request);

        if (!$request->getCookie('landing_page')) {
            $expiry = new Time('+ 1 year');
            $response = $response->withCookie(new Cookie(
                'landing_page',
                $request->getRequestTarget(),
                $expiry
            ));
        }
```

(continues on next page)

(continuação da página anterior)

```
        return $response;
    }
}
```

Agora que criamos um middleware muito simples, vamos anexá-lo ao nosso aplicativo:

```
// Em src/Application.php
namespace App;

use App\Middleware\TrackingCookieMiddleware;
use Cake\Http\MiddlewareQueue;

class Application
{
    public function middleware(MiddlewareQueue $middlwareQueue): MiddlewareQueue
    {
        // Adicione seu middleware simples à fila
        $middlwareQueue->add(new TrackingCookieMiddleware());

        // Adicione um pouco mais de middleware à fila

        return $middlwareQueue;
    }
}
```

Roteamento de Middleware

O middleware de roteamento é responsável por aplicar as rotas no seu aplicativo e resolver o: plug-in, o controlador e a ação que uma solicitação está pedindo. Ele pode armazenar em cache a coleção de rotas usada no seu aplicativo para aumentar o tempo de inicialização. Para habilitar o cache de rotas em, forneça o *cache configuration* desejado como um parâmetro:

```
// Em Application.php
public function middleware(MiddlewareQueue $middlwareQueue): MiddlewareQueue
{
    // ...
    $middlwareQueue->add(new RoutingMiddleware($this, 'routing'));
}
```

O exemplo acima usaria o mecanismo de cache `routing` para armazenar a coleção de rotas gerada.

Middleware de Cabeçalho de Segurança

A camada `Security Headers Middleware` facilita a aplicação de cabeçalhos relacionados à segurança em seu aplicativo. Depois de configurado, o middleware pode aplicar os seguintes cabeçalhos às respostas:

- `X-Content-Type-Options`
- `X-Download-Options`
- `X-Frame-Options`
- `X-Permitted-Cross-Domain-Policies`

- `Referrer-Policy`

Esse middleware é configurado usando uma interface simples antes de ser aplicado à pilha de middleware do seu aplicativo:

```
use Cake\Http\Middleware\SecurityHeadersMiddleware;

$securityHeaders = new SecurityHeadersMiddleware();
$securityHeaders
    ->setCrossDomainPolicy()
    ->setReferrerPolicy()
    ->setXFrameOptions()
    ->setXssProtection()
    ->noOpen()
    ->noSniff();

$middlwareQueue->add($securityHeaders);
```

Middleware do Cabeçalho da Política de Segurança de Conteúdo

O `CspMiddleware` facilita a adição de cabeçalhos referente a política de segurança de conteúdo em seu aplicativo. Antes de usá-lo, você deve instalar o `paragonie/csp-builder`:

Você pode configurar o middleware usando uma matriz ou passando um objeto `CSPBuilder` integrado:

```
use Cake\Http\Middleware\CspMiddleware;

$csp = new CspMiddleware([
    'script-src' => [
        'allow' => [
            'https://www.google-analytics.com',
        ],
        'self' => true,
        'unsafe-inline' => false,
        'unsafe-eval' => false,
    ],
]);

$middlewareQueue->add($csp);
```

Middleware de Cookie Criptografado

Se o seu aplicativo possui cookies que contêm dados que você deseja ofuscar e proteger contra adulterações do usuário, você pode usar o middleware de cookies criptografado do CakePHP para criptografar e descriptografar de forma transparente os dados de cookies via middleware. Os dados dos cookies são criptografados via OpenSSL usando AES:

```
use Cake\Http\Middleware\EncryptedCookieMiddleware;

$cookies = new EncryptedCookieMiddleware(
    // Nomes de cookies para proteção
    ['secrets', 'protected'],
    Configure::read('Security.cookieKey')
```

(continues on next page)

(continuação da página anterior)

```
);

$middlwareQueue->add($cookies);
```

Nota: É recomendável que a chave de criptografia usada para os dados do cookie seja usada *exclusivamente* para os dados do cookie.

Os algoritmos de criptografia e o estilo de preenchimento usados pelo middleware do cookie são compatíveis com o `CookieComponent` de versões anteriores do CakePHP.

Falsificação de Solicitação entre Sites (CSRF) Middleware

A proteção CSRF pode ser aplicada a todo o aplicativo ou a escopos de roteamento específicos.

Nota: Você não pode usar as duas abordagens a seguir juntas; deve escolher apenas uma. Se você usar as duas abordagens juntas, ocorrerá um erro de incompatibilidade de token CSRF em cada solicitação *PUT* e` POST`

Ao aplicar o `CsrfProtectionMiddleware` à pilha de middleware do Aplicativo, você protege todas as ações no aplicativo:

```
// Em src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;

public function middleware($middlwareQueue) {
    $options = [
        // ...
    ];
    $csrf = new CsrfProtectionMiddleware($options);

    $middlwareQueue->add($csrf);

    return $middlwareQueue;
}
```

Ao aplicar o `CsrfProtectionMiddleware` aos escopos de roteamento, você pode incluir ou excluir grupos de rotas específicos:

```
// Em src/Application.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;

public function routes($routes) {
    $options = [
        // ...
    ];
    $routes->registerMiddleware('csrf', new CsrfProtectionMiddleware($options));
    parent::routes($routes);
}

// Em config/routes.php
```

(continues on next page)

(continuação da página anterior)

```
Router::scope('/', function (RouteBuilder $routes) {
    $routes->applyMiddleware('csrf');
});
```

As opções podem ser passadas para o construtor do middleware. As opções de configuração disponíveis são:

- `cookieName` O nome do cookie a ser enviado. O padrão é `` csrfToken``.
- `expiry` Quanto tempo o token CSRF deve durar. O padrão é a sessão do navegador.
- `secure` Se o cookie será ou não definido com o sinalizador Secure. Isso é, o cookie será definido apenas em uma conexão HTTPS e qualquer tentativa no HTTP normal falhará. O padrão é `false`.
- `httpOnly` Se o cookie será ou não definido com o sinalizador HttpOnly. O padrão é `false`.
- `field` O campo do formulário a ser verificado. O padrão é `_csrfToken`. Alterar isso também exigirá a configuração do FormHelper.

Quando ativado, você pode acessar o token CSRF atual no objeto de solicitação:

```
$token = $this->request->getParam('_csrfToken');
```

Nota: Você deve aplicar o middleware de proteção CSRF apenas para URLs que manipulam solicitações com estado usando cookies/sessão. Solicitações sem estado, por ex. ao desenvolver uma API, não são afetados pelo CSRF; portanto, o middleware não precisa ser aplicado a essas URLs.

Integração com FormHelper

O `CsrfProtectionMiddleware` se integra perfeitamente ao `FormHelper`. Cada vez que você cria um formulário com `FormHelper`, ele insere um campo oculto que contém o token CSRF.

Nota: Ao usar a proteção CSRF, você sempre deve iniciar seus formulários com o `FormHelper`. Caso contrário, será necessário criar manualmente entradas ocultas em cada um dos seus formulários.

Solicitações de Proteção CSRF e AJAX

Além de solicitar parâmetros de dados, os tokens CSRF podem ser enviados por meio de um cabeçalho especial `X-CSRF-Token`. O uso de um cabeçalho geralmente facilita a integração de um token CSRF com aplicativos pesados de JavaScript ou endpoints de API baseados em XML/JSON.

O token CSRF pode ser obtido através do cookie `csrfToken`.

Body Parser Middleware

Se seu aplicativo aceitar JSON, XML ou outros corpos de solicitação codificados, o `BodyParserMiddleware` permitirá que você decodifique essas solicitações em uma matriz que esteja disponível em `$request->getParsedData()` e `$request->getData()`. Por padrão, apenas os corpos `json` serão analisados, mas a análise XML pode ser ativada com uma opção. Você também pode definir seus próprios analisadores:

```
use Cake\Http\Middleware\BodyParserMiddleware;

// somente JSON será analisado.
$bodies = new BodyParserMiddleware();

// Ativar análise XML
$bodies = new BodyParserMiddleware(['xml' => true]);

// Desativar a análise JSON
$bodies = new BodyParserMiddleware(['json' => false]);

// Adicione seu próprio analisador que corresponda aos
// valores do cabeçalho do tipo de conteúdo à chamada que pode analisá-los.
$bodies = new BodyParserMiddleware();
$bodies->addParser(['text/csv'], function ($body, $request) {
    // Use uma biblioteca de análise CSV.
    return Csv::parse($body);
});
```

CAPÍTULO 27

Sessões

O CakePHP fornece um wrapper e um conjunto de recursos de utilitários sobre a extensão `session` nativa do PHP. As sessões permitem identificar usuários únicos em solicitações e armazenar dados persistentes para usuários específicos. Ao contrário dos cookies, os dados da sessão não estão disponíveis no lado do cliente. O uso de `$_SESSION` geralmente é evitado no CakePHP, e o uso das classes Session é preferido.

Configuração da Sessão

A configuração da sessão é geralmente definida em `/config/app.php`. As opções disponíveis são:

- `Session.timeout` - O número de *minutos* antes que o manipulador de sessões do CakePHP expire a sessão
- `Session.defaults` - Permite usar as configurações de sessão padrão incorporadas como base para sua configuração de sessão. Veja abaixo os padrões internos.
- `Session.handler` - Permite definir um manipulador de sessão personalizado. O banco de dados principal e os manipuladores de sessão de cache usam isso. Veja abaixo informações adicionais sobre manipuladores de sessão.
- `Session.ini` - Permite definir configurações adicionais de sessão ini para sua configuração. Isso combinado com `Session.handler` substitui os recursos de manipulação de sessão personalizados das versões anteriores
- `Session.cookie` - O nome do cookie em uso, o padrão é "CAKEPHP".
- `Session.cookiePath` - O caminho da URL para o qual o cookie de sessão está definido. Mapeia para a configuração php.ini `session.cookie_path`. O padrão é o caminho base do aplicativo.

O padrão do CakePHP `session.cookie_secure` é `true`, quando seu aplicativo está em um protocolo SSL. Se seu aplicativo atender a partir de protocolos SSL e não SSL, você poderá ter problemas com a perda de sessões. Se você precisar acessar a sessão nos domínios SSL e não SSL, desabilite isso:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_secure' => false
    ]
]);
```

O caminho do cookie da sessão é padronizado como o caminho base do aplicativo. Para mudar isso, você pode usar o valor ini `session.cookie_path`. Por exemplo, se você deseja que sua sessão persista em todos os subdomínios, você pode:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_path' => '/',
        'session.cookie_domain' => '.yourdomain.com'
    ]
]);
```

Por padrão, o PHP define o cookie da sessão para expirar assim que o navegador é fechado, independentemente do valor configurado `Session.timeout`. O tempo limite do cookie é controlado pelo valor ini `session.cookie_lifetime` e pode ser configurado usando:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        // Invalide o cookie após 30 minutos sem visitar
        // qualquer página do site.
        'session.cookie_lifetime' => 1800
    ]
]);
```

A diferença entre `Session.timeout` e o valor `session.cookie_lifetime` é que este último depende do cliente dizer a verdade sobre o cookie. Se você precisar de uma verificação de tempo limite mais rigorosa, sem depender do que o cliente relata, use `Session.timeout`.

Observe que `Session.timeout` corresponde ao tempo total de inatividade para um usuário (ou seja, o tempo sem visitar nenhuma página em que a sessão é usada) e não limita a quantidade total de minutos que um usuário pode permanecer no site.

Manipuladores de sessão e configuração incorporados

O CakePHP vem com várias configurações de sessão embutidas. Você pode usá-los como base para a configuração da sessão ou criar uma solução totalmente personalizada. Para usar padrões, basta definir a chave “defaults” como o nome do padrão que você deseja usar. Você pode substituir qualquer subconjunto declarando-o na sua configuração de sessão:

```
Configure::write('Session', [
    'defaults' => 'php'
]);
```

O exemplo acima irá usar a configuração de sessão “php” embutida. Você pode aumentar parte ou a totalidade fazendo o seguinte:

```
Configure::write('Session', [
    'defaults' => 'php',
    'cookie' => 'my_app',
    'timeout' => 4320 // 3 dias
]);
```

O texto acima substitui o tempo limite e o nome do cookie para a configuração da sessão "php". As configurações internas são:

- `php` - Salva sessões com as configurações padrão no seu arquivo php.ini.
- `cake` - Salva sessões como arquivos dentro de `tmp/sessions`. Essa é uma boa opção quando em hosts que não permitem que você escreva fora de seu próprio diretório.
- `database` - Use as sessões de banco de dados internas. Veja abaixo para mais informações.
- `cache` - Use as sessões de cache internas. Veja abaixo para mais informações.

Manipuladores de Sessão

Os manipuladores de sessão também podem ser definidos na matriz de configuração da sessão. Ao definir a chave de configuração "handler.engine", você pode nomear a classe ou fornecer uma instância do manipulador. A classe/objeto deve implementar o PHP nativo `SessionHandlerInterface`. A implementação dessa interface permitirá que a `Session` mapeie automaticamente os métodos para o manipulador. Os principais manipuladores de sessão do Cache e do Banco de Dados usam esse método para salvar sessões. Configurações adicionais para o manipulador devem ser colocadas dentro da matriz do manipulador. Você pode então ler esses valores de dentro do seu manipulador:

```
'Session' => [
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions'
    ]
]
```

A amostra acima, exemplifica como você pode configurar o manipulador de sessões do banco de dados com um modelo de aplicativo. Ao usar nomes de classe como seu handler.engine, o CakePHP espera encontrar sua classe no namespace `Http\Session`. Por exemplo, se você tiver uma classe `AppSessionHandler`, o arquivo deve ser **src/Http/Session/AppSessionHandler.php** e o nome da classe deve ser `App\Http\Session\AppSessionHandler`. Você também pode usar manipuladores de sessão de plugins internos. Configurando o mecanismo para `MyPlugin.PluginSessionHandler`.

Nota: Antes da versão 3.6.0, os arquivos do adaptador de sessão devem ser colocados em **src/Network/Session/AppHandler.php**.

Sessões de Banco de Dados

Se você precisar usar um banco de dados para armazenar os dados da sessão, configure da seguinte maneira:

```
'Session' => [
    'defaults' => 'database'
]
```

Essa configuração requer uma tabela de banco de dados, com este esquema:

```
CREATE TABLE `sessions` (
  `id` char(40) CHARACTER SET ascii COLLATE ascii_bin NOT NULL,
  `created` datetime DEFAULT CURRENT_TIMESTAMP, -- Optional
  `modified` datetime DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, -- Optional
  `data` blob DEFAULT NULL, -- for PostgreSQL use bytea instead of blob
  `expires` int(10) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Você pode encontrar uma cópia do esquema para a tabela de sessões no esqueleto do aplicativo[124] em `config/schema/sessions.sql`.

Você também pode usar sua própria classe `Tabela` para lidar com o salvamento das sessões:

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        'engine' => 'DatabaseSession',
        'model' => 'CustomSessions'
    ]
]
```

O comando acima instruirá a Sessão a usar os padrões internos do "banco de dados" e especificará que uma Tabela chamada `CustomSessions` será a escolhida para salvar as informações da sessão no banco de dados.

Sessões de Cache

A classe Cache também pode ser usada para armazenar sessões. Isso permite que você armazene sessões em um cache como APCu ou Memcached. Existem algumas ressalvas no uso de sessões de cache, pois se você esgotar o espaço em cache, as sessões começarão a expirar à medida que os registros forem despejados.

Para usar sessões baseadas em cache, você pode configurar sua configuração de sessão como:

```
Configure::write('Session', [
    'defaults' => 'cache',
    'handler' => [
        'config' => 'session'
    ]
]);
```

Isso configurará a Session para usar a classe `CacheSession` como o delegado para salvar as sessões. Você pode usar a chave "config" para configuração de uso do cache. A configuração padrão do cache é `'default'`.

[124] https://github.com/cakephp/app

Definindo diretivas ini

Os padrões internos tentam fornecer uma base comum para a configuração da sessão. Pode ser necessário ajustar também sinalizadores ini específicos. O CakePHP expõe a capacidade de personalizar as configurações ini para as configurações padrão e personalizadas. A chave `ini` nas configurações da sessão permite especificar valores de configuração individuais. Por exemplo, você pode usá-lo para controlar configurações como `session.gc_divisor`:

```
Configure::write('Session', [
    'defaults' => 'php',
    'ini' => [
        'session.cookie_name' => 'MyCookie',
        'session.cookie_lifetime' => 1800, // Valid for 30 minutes
        'session.gc_divisor' => 1000,
        'session.cookie_httponly' => true
    ]
]);
```

Criando um manipulador de sessão personalizado

Criar um manipulador de sessão personalizado é simples no CakePHP. Neste exemplo, criaremos um manipulador de sessão que armazena sessões no cache (APC) e no banco de dados. Isso nos dá o melhor das E/S rápidas da APC, sem a necessidade de se preocupar com a evaporação das sessões quando o cache ficar cheio.

Primeiro, precisamos criar nossa classe personalizada e colocá-la em **src/Http/Session/ComboSession.php**. A classe deve se parecer com:

```
namespace App\Http\Session;

use Cake\Cache\Cache;
use Cake\Core\Configure;
use Cake\Http\Session\DatabaseSession;

class ComboSession extends DatabaseSession
{
    public $cacheKey;

    public function __construct()
    {
        $this->cacheKey = Configure::read('Session.handler.cache');
        parent::__construct();
    }

    // Lê dados da sessão.
    public function read($id)
    {
        $result = Cache::read($id, $this->cacheKey);
        if ($result) {
            return $result;
        }

        return parent::read($id);
```

(continues on next page)

(continuação da página anterior)

```
    }

    // Gravar dados na sessão.
    public function write($id, $data)
    {
        Cache::write($id, $data, $this->cacheKey);

        return parent::write($id, $data);
    }

    // Apaga uma sessão.
    public function destroy($id)
    {
        Cache::delete($id, $this->cacheKey);

        return parent::destroy($id);
    }

    // Remove sessões expiradas.
    public function gc($expires = null)
    {
        return Cache::gc($this->cacheKey) && parent::gc($expires);
    }
}
```

Nossa classe estende o `DatabaseSession` interno, para que não tenhamos que duplicar toda a sua lógica e comportamento. Envolvemos cada operação com uma operação *Cake\Cache\Cache*. Isso nos permite buscar sessões no cache rápido e não ter que nos preocupar com o que acontece quando o cache é preenchido. Usar este manipulador de sessões também é fácil. No seu **app.php**, faça com que o bloco de sessões esteja como o seguinte:

```
'Session' => [
    'defaults' => 'database',
    'handler' => [
        'engine' => 'ComboSession',
        'model' => 'Session',
        'cache' => 'apc'
    ]
],
// Certifique-se de adicionar uma configuração de cache apc
'Cache' => [
    'apc' => ['engine' => 'Apc']
]
```

Agora, nosso aplicativo começará a usar nosso manipulador de sessão personalizado para ler e gravar dados da sessão.

class Session

Acessando o Objeto de Sessão

Você pode acessar os dados da sessão em qualquer lugar em que tenha acesso a um objeto de solicitação. Isso significa que a sessão é acessível em:

- Controllers
- Views
- Helpers
- Cells
- Components

Além do objeto básico da sessão, você também pode usar o `Cake\View\Helper\SessionHelper` para interagir com a sessão nas suas visualizações. Um exemplo básico de uso da sessão seria:

```
// Antes da versão 3.6.0, use session()
$name = $this->getRequest()->getSession()->read('User.name');

// Se você estiver acessando a sessão várias vezes,
// provavelmente desejará uma variável local.
$session = $this->getRequest()->getSession();
$name = $session->read('User.name');
```

Leitura e gravação de dados da sessão

`Session::read`(*$key*)

Você pode ler valores da sessão usando `Hash::extract()`:

```
$session->read('Config.language');
```

`Session::write`(*$key*, *$value*)

`$key` deve ser o caminho separado por pontos que você deseja escrever `$value` para:

$session->write(“Config.language”, “en”);

Você também pode especificar um ou vários hashes assim:

```
$session->write([
  'Config.theme' => 'blue',
  'Config.language' => 'en',
]);
```

`Session::delete`(*$key*)

Quando você precisar excluir dados da sessão, poderá usar `delete()`:

```
$session->delete('Some.value');
```

`static Session::consume`(*$key*)

Quando você precisar ler e excluir dados da sessão, poderá usar `consume()`:

```
$session->consume('Some.value');
```

Session::**check**(*$key*)

Se você deseja ver se existem dados na sessão, você pode usar `check()`:

```
if ($session->check('Config.language')) {
    // Config.language exists existe e não é nulo.
}
```

Destruindo a Sessão

Session::**destroy**()

Destruir a sessão é útil quando os usuários efetuam logout. Para destruir uma sessão, use o método `destroy()`:

```
$session->destroy();
```

Destruir uma sessão removerá todos os dados do servidor na sessão, mas **não** removerá o cookie da sessão.

Identificadores de Sessão Rotativos

Session::**renew**()

Embora o `AuthComponent` renove automaticamente o ID da sessão quando os usuários se conectam e se desconectam, pode ser necessário girar os IDs da sessão manualmente. Para fazer isso, use o método `renew()`:

```
$session->renew();
```

Mensagens em Flash

Flash messages are small messages displayed to end users once. They are often used to present error messages, or confirm that actions took place successfully.

To set and display flash messages you should use *Flash* and *Flash*

Mensagens em Flash são pequenas mensagens exibidas para os usuários finais uma vez. Eles são frequentemente usados para apresentar mensagens de erro ou confirmar que as ações foram realizadas com êxito.

Para definir e exibir mensagens em flash, você deve usar *Flash* e *Flash*.

CAPÍTULO 28

Testing

O CakePHP vem com suporte interno para testes e integração para o PHPUnit[125]. 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[126] ou Composer[127].

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
```

[125] https://phpunit.de
[126] https://phpunit.de/#download
[127] https://getcomposer.org

Usando o arquivo PHAR

Depois de ter baixado o arquivo **phpunit.phar** , você pode usar ele para executar seus testes:

```
php phpunit.phar
```

Dica: Como conveniência você pode deixar phpunit.phar disponivel 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 Windows[128].

Configuração do banco de dados de teste

Lembre-se de ter o debug habilitado em seu arquivo **config/app.php** antes de executar qualquer teste. Antes de executar quaisquer testes você deve adicionar um datasourse `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'
    ],
],
```

Nota: É uma boa idéia tornar o banco de dados de teste e o banco de dados real diferentes. Isso evitará erros embaraçosos mais tarde.

Verificando a Configuração de Teste

Depois de instalar o PHPUnit e definir a configuração da fonte de dados `test`, você pode se certificar de que está pronto para escrever e executar seus próprios testes executando os testes do aplicativo:

```
# Para phpunit.phar
$ php phpunit.phar

# Para phpunit instalado com Composer
$ vendor/bin/phpunit
```

[128] https://phpunit.de/manual/current/en/installation.html#installation.phar.windows

O exemplo acima deve executar todos os testes que você possui ou informar que nenhum teste foi executado. Para executar um teste específico, você pode fornecer o caminho para o teste como um parâmetro para o PHPUnit. Por exemplo, se você tiver um caso de teste para a classe ArticlesTable, poderá executá-lo com

```
$ vendor/bin/phpunit tests/TestCase/Model/Table/ArticlesTableTest
```

Você deve ver uma barra verde com algumas informações adicionais sobre os testes executados e o número passado.

Nota: Se você estiver em um sistema Windows, provavelmente não verá cores.

Convenções de Casos de Teste

Como a maioria das coisas no CakePHP, os casos de teste têm algumas convenções. No que diz respeito aos testes:

1. Os arquivos PHP que contêm testes devem estar nos seus diretórios `tests/TestCase/[Type]`.
2. Os nomes desses arquivos devem terminar em **Test.php** em vez de apenas em .php.
3. As classes que contêm testes devem se estender das classes `Cake\TestSuite\TestCase`, `Cake\TestSuite\IntegrationTestCase` ou `\PHPUnit\Framework\TestCase`.
4. Como outros nomes de classe, os nomes de classe do caso de teste devem corresponder ao nome do arquivo. **RouterTest.php** deve conter `classe RouterTest extends TestCase`.
5. O nome de qualquer método que contenha um teste (ou seja, que contenha uma asserção) deve começar com `test`, como em `testPublished()`. Você também pode usar a anotação `@test` para marcar métodos como métodos de teste.

Criando seu Primeiro Caso de Teste

No exemplo a seguir, criaremos um caso de teste para um método auxiliar muito simples. O auxiliar que vamos testar estará formatando a barra de progresso HTML. Nosso ajudante se parece com:

```
namespace App\View\Helper;

use Cake\View\Helper;

class ProgressHelper extends Helper
{
    public function bar($value)
    {
        $width = round($value / 100, 2) * 100;

        return sprintf(
            '<div class="progress-container">
                <div class="progress-bar" style="width: %s%%"></div>
            </div>', $width);
    }
}
```

Este é um exemplo muito simples, mas será útil mostrar como você pode criar um caso de teste simples. Após criar e salvar nosso auxiliar, criaremos o arquivo de caso de teste em **tests/TestCase/View/Helper/ProgressHelperTest.php**. Nesse arquivo começaremos com o seguinte:

```
namespace App\Test\TestCase\View\Helper;

use App\View\Helper\ProgressHelper;
use Cake\TestSuite\TestCase;
use Cake\View\View;

class ProgressHelperTest extends TestCase
{
    public function setUp()
    {

    }

    public function testBar()
    {

    }
}
```

Vamos preparar esse esqueleto em um minuto. Adicionamos dois métodos para começar. Primeiro é `setUp()`. Este método é chamado antes de cada método *test* em uma classe de caso de teste. Os métodos de instalação devem inicializar os objetos necessários para o teste e fazer qualquer configuração necessária. No nosso método de configuração, adicionaremos o seguinte:

```
public function setUp()
{
    parent::setUp();
    $View = new View();
    $this->Progress = new ProgressHelper($View);
}
```

A chamada do método pai é importante nos casos de teste, pois `TestCase::setUp()` faz várias coisas, como fazer backup dos valores em *Configure* e, armazenar os caminhos em *App*.

Em seguida, preencheremos o método de teste. Usaremos algumas asserções para garantir que nosso código crie a saída que esperamos:

```
public function testBar()
{
    $result = $this->Progress->bar(90);
    $this->assertContains('width: 90%', $result);
    $this->assertContains('progress-bar', $result);

    $result = $this->Progress->bar(33.3333333);
    $this->assertContains('width: 33%', $result);
}
```

O teste acima é simples, mas mostra o benefício potencial do uso de casos de teste. Usamos `assertContains()` para garantir que nosso assistente retorne uma string que contenha o conteúdo que esperamos. Se o resultado não contiver o conteúdo esperado, o teste falhará e saberemos que nosso código está incorreto.

Usando casos de teste, você pode descrever o relacionamento entre um conjunto de entradas conhecidas e sua saída

esperada. Isso ajuda você a ter mais confiança no código que está escrevendo, pois pode garantir que o código que você escreveu atenda às expectativas e afirmações feitas pelos seus testes. Além disso, como os testes são de código, eles são fáceis de executar novamente sempre que você faz uma alteração. Isso ajuda a impedir a criação de novos bugs.

Nota: O EventManager é atualizado para cada método de teste. Isso significa que, ao executar vários testes ao mesmo tempo, você perderá seus ouvintes de eventos que foram registrados no config/bootstrap.php, pois o bootstrap é executado apenas uma vez.

Executando Testes

Depois de instalar o PHPUnit e escrever alguns casos de teste, você deverá executá-los com muita frequência. É uma boa ideia executar testes antes de confirmar quaisquer alterações para ajudar a garantir que você não tenha quebrado nada.

Ao usar o `phpunit`, você pode executar os testes do aplicativo. Para executar os testes do seu aplicativo, você pode simplesmente executar:

```
# instalado pelo Composer
$ vendor/bin/phpunit

# arquivo phar
php phpunit.phar
```

Se você clonou o código-fonte *CakePHP do GitHub <https://github.com/cakephp/cakephp>* __ e deseja executar os testes de unidade do CakePHP, não se esqueça de executar o seguinte comando `Composer` antes de executar `phpunit` para que todas as dependências sejam instaladas:

```
$ composer install
```

No diretório raiz do seu aplicativo. Para executar testes para um plug-in que faz parte da fonte do aplicativo, primeiro execute `cd` para o diretório do plug-in, depois use o comando `phpunit` que corresponde à maneira como você instalou o phpunit:

```
cd plugins

# Usando o phpunit instalado pelo compositor
../vendor/bin/phpunit

# Usando o arquivo phar
php ../phpunit.phar
```

Para executar testes em um plug-in independente, você deve primeiro instalar o projeto em um diretório separado e instalar suas dependências:

```
git clone git://github.com/cakephp/debug_kit.git
cd debug_kit
php ~/composer.phar install
php ~/phpunit.phar
```

Filtrando Casos de Teste

Quando você tem casos de teste maiores, geralmente deseja executar um subconjunto dos métodos de teste ao tentar trabalhar em um único caso com falha. Com o corredor da CLI, você pode usar uma opção para filtrar os métodos de teste:

```
$ phpunit --filter testSave tests/TestCase/Model/Table/ArticlesTableTest
```

O parâmetro filter é usado como uma expressão regular com distinção entre maiúsculas e minúsculas para filtrar quais métodos de teste executar.

Gerando Cobertura de Código

Você pode gerar relatórios de amostras de código a partir da linha de comando usando as ferramentas internas de cobertura de código do PHPUnit. O PHPUnit irá gerar um conjunto de arquivos HTML estáticos contendo os resultados da cobertura. Você pode gerar cobertura para um caso de teste, fazendo o seguinte:

```
$ phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/ArticlesTableTest
```

Isso colocará os resultados da cobertura no diretório webroot do seu aplicativo. Você deve conseguir visualizar os resultados acessando `http://localhost/your_app/coverage`.

Se você estiver usando o PHP 5.6.0 ou superior, poderá usar o `phpdbg` para gerar cobertura em vez do xdebug. O `phpdbg` geralmente é mais rápido na geração de cobertura:

```
$ phpdbg -qrr phpunit --coverage-html webroot/coverage tests/TestCase/Model/Table/
↪ArticlesTableTest
```

Combinando Conjuntos de Testes para Plug-ins

Muitas vezes, seu aplicativo será composto de vários plugins. Nessas situações, pode ser bastante entediante executar testes para cada plug-in. Você pode fazer testes em execução para cada um dos plugins que compõem seu aplicativo adicionando seções adicionais `<testsuite>` ao arquivo **phpunit.xml.dist** do seu aplicativo:

```
<testsuites>
    <testsuite name="app">
        <directory>./tests/TestCase/</directory>
    </testsuite>

    <!-- Adicione seus pacotes de plugins -->
    <testsuite name="forum">
        <directory>./plugins/Forum/tests/TestCase/</directory>
    </testsuite>
</testsuites>
```

Quaisquer suítes de teste adicionais vinculados ao elemento `<testsuites>` serão executados automaticamente quando você usar `phpunit`.

Se você estiver usando `<testsuites>` para usar acessórios de plug-ins que você instalou com o composer, o arquivo `composer.json` do plugin deve adicionar o espaço para nome do fixture à seção de carregamento automático. Exemplo:

```
"autoload-dev": {
    "psr-4": {
        "PluginName\\Test\\Fixture\\": "tests/Fixture/"
    }
},
```

Retornos de Chamada do Ciclo de Cida do Caso de Teste

Os casos de teste têm vários retornos de chamada do ciclo de vida que você pode usar ao fazer o teste:

- `setUp` é chamado antes de cada método de teste. Deve ser usado para criar os objetos que serão testados e inicializar quaisquer dados para o teste. Lembre-se sempre de chamar `parent::setUp()`
- `tearDown` é chamado após cada método de teste. Deve ser usado para limpeza após a conclusão do teste. Lembre-se sempre de chamar `parent::tearDown()`.
- `setupBeforeClass` é chamado uma vez antes do início dos métodos de teste em um caso. Este método deve ser *estático*.
- `tearDownAfterClass` é chamado uma vez depois que os métodos de teste em um caso são iniciados. Este método deve ser *estático*.

Fixtures

Ao testar o código que depende dos modelos e do banco de dados, pode-se usar **fixtures** como uma maneira de gerar tabelas de dados temporárias carregadas com dados de amostra que podem ser usados pelo teste. O benefício do uso de fixtures é que seu teste não tem chance de interromper os dados do aplicativo ao vivo. Além disso, você pode começar a testar seu código antes de realmente desenvolver conteúdo ao vivo para um aplicativo.

O CakePHP usa a conexão chamada `test` no seu arquivo de configuração **config/app.php**. Se essa conexão não for utilizável, uma exceção será gerada e você não poderá usar fixtures de banco de dados.

O CakePHP executa o seguinte durante o curso de uma fixture no caso de teste:

1. Cria tabelas para cada um dos fixtures necessários.
2. Preenche tabelas com dados, se os dados forem fornecidos no fixture.
3. Executa métodos de teste.
4. Esvazia as tabelas de fixtures.
5. Remove tabelas de fixtures do banco de dados.

Conexões de Teste

Por padrão, o CakePHP fará o pseudônimo de cada conexão em sua aplicação. Cada conexão definida no bootstrap do seu aplicativo que não inicia com `test` terá um alias prefixado criado com `test`. As conexões com aliasing garantem que você não use acidentalmente a conexão errada nos casos de teste. O alias de conexão é transparente para o restante do seu aplicativo. Por exemplo, se você usar a conexão "padrão", receberá a conexão `test` nos casos de teste. Se você usar a conexão "replica", o conjunto de testes tentará usar "test_replica".

Criando Fixtures

Ao criar um dispositivo elétrico, você definirá principalmente duas coisas: como a tabela é criada (quais campos fazem parte da tabela) e quais registros serão preenchidos inicialmente na tabela. Vamos criar nosso primeiro fixture, que será usado para testar nosso próprio modelo de artigo. Crie um arquivo chamado **ArticlesFixture.php** no seu diretório **tests/Fixture**, com o seguinte conteúdo:

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{
      // Opcional. Configure esta propriedade para carregar fixtures
      // em uma fonte de dados de teste diferente
      public $connection = 'test';

      public $fields = [
          'id' => ['type' => 'integer'],
          'title' => ['type' => 'string', 'length' => 255, 'null' => false],
          'body' => 'text',
          'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
          'created' => 'datetime',
          'modified' => 'datetime',
          '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']]
          ]
      ];
      public $records = [
          [
              'title' => 'First Article',
              'body' => 'First Article Body',
              'published' => '1',
              'created' => '2007-03-18 10:39:23',
              'modified' => '2007-03-18 10:41:31'
          ],
          [
              'title' => 'Second Article',
              'body' => 'Second Article Body',
              'published' => '1',
              'created' => '2007-03-18 10:41:23',
              'modified' => '2007-03-18 10:43:31'
          ],
          [
              'title' => 'Third Article',
              'body' => 'Third Article Body',
              'published' => '1',
              'created' => '2007-03-18 10:43:23',
              'modified' => '2007-03-18 10:45:31'
          ]
      ];
 }
```

Nota: Recomenda-se não adicionar valores manualmente a colunas incrementais automáticas, pois isso interfere na geração de sequência no PostgreSQL e SQLServer.

A propriedade `$connection` define a fonte de dados que a fixture usará. Se seu aplicativo usa várias fontes de dados, você deve fazer com que as fixtures correspondam às fontes de dados do modelo, mas prefixados com `test`. Por exemplo, se o seu modelo usa a fonte de dados `mydb`, sua fixture deve usar a fonte de dados `test_mydb`. Se a conexão `test_mydb` não existir, seus modelos usarão a fonte de dados `test` padrão. As fontes de dados da fixture devem ser prefixadas com `test` para reduzir a possibilidade de truncar acidentalmente todos os dados do seu aplicativo ao executar testes.

Usamos `$fields` para especificar quais campos farão parte desta tabela e como eles são definidos. O formato usado para definir esses campos é o mesmo usado com `Cake\Database\Schema\Table`. As chaves disponíveis para definição da tabela são:

type
Tipo de dados interno do CakePHP. Atualmente suportado:

- `string`: mapeia para `VARCHAR` ou `CHAR`
- `uuid`: mapeia para `UUID`
- `text`: mapeia para `TEXT`
- `integer`: mapeia para `INT`
- `biginteger`: mapeia para `BIGINTEGER`
- `decimal`: mapeia para `DECIMAL`
- `float`: mapeia para `FLOAT`
- `datetime`: mapeia para `DATETIME`
- `timestamp`: mapeia para `TIMESTAMP`
- `time`: mapeia para `TIME`
- `date`: mapeia para `DATE`
- `binary`: mapeia para `BLOB`

fixed
Usado com tipos de sequência para criar colunas CHAR em plataformas que as suportam.

length
Defina para o comprimento específico que o campo deve ter.

precision
Defina o número de casas decimais usadas nos campos flutuante e decimal.

null
Defina como `true` (para permitir NULLs) ou `false` (para desabilitar NULLs).

default
Valor padrão que o campo assume.

Podemos definir um conjunto de registros que serão preenchidos após a criação da tabela de fixtures. O formato é bastante simples, `$records` é uma matriz de registros. Cada item em `$records` deve ser uma única linha. Dentro de cada linha, deve haver uma matriz associativa das colunas e valores para a linha. Lembre-se de que cada registro na matriz $records deve ter uma chave para **todos** os campos especificados na matriz `$fields`. Se um campo para um registro específico precisar ter um valor `null`, basta especificar o valor dessa chave como `null`.

Dados Dinâmicos e Fixtures

Como os registros de uma fixture são declarados como uma propriedade de classe, você não pode usar funções ou outros dados dinâmicos para definir fixtures. Para resolver esse problema, você pode definir `$records` na função `init()` de sua fixture. Por exemplo, se você quiser que todos os carimbos de data e hora criados e modificados reflitam a data de hoje, faça o seguinte:

```
namespace App\Test\Fixture;

use Cake\TestSuite\Fixture\TestFixture;

class ArticlesFixture extends TestFixture
{
    public $fields = [
        'id' => ['type' => 'integer'],
        'title' => ['type' => 'string', 'length' => 255, 'null' => false],
        'body' => 'text',
        'published' => ['type' => 'integer', 'default' => '0', 'null' => false],
        'created' => 'datetime',
        'modified' => 'datetime',
        '_constraints' => [
            'primary' => ['type' => 'primary', 'columns' => ['id']],
        ]
    ];

    public function init()
    {
        $this->records = [
            [
                'title' => 'First Article',
                'body' => 'First Article Body',
                'published' => '1',
                'created' => date('Y-m-d H:i:s'),
                'modified' => date('Y-m-d H:i:s'),
            ],
        ];
        parent::init();
    }
}
```

Ao substituir `init()` lembre-se de sempre chamar `parent::init()`.

Importando Informações da Tabela

Definir o esquema nos arquivos de fixture pode ser realmente útil ao criar plug-ins ou bibliotecas se você estiver criando um aplicativo que precise ser portátil entre os fornecedores de banco de dados. Redefinir o esquema em acessórios pode se tornar difícil de manter em aplicativos maiores. Devido a isso, o CakePHP fornece a capacidade de importar o esquema de uma conexão existente e usar a definição de tabela refletida para criar a definição de tabela usada no conjunto de testes.

Vamos começar com um exemplo. Supondo que você tenha uma tabela com os artigos disponíveis no seu aplicativo, altere o exemplo de dispositivo fornecido na seção anterior (**tests/Fixture/ArticlesFixture.php**) para:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
}
```

Se você deseja usar uma conexão diferente, use:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles', 'connection' => 'other'];
}
```

Normalmente, você também tem uma classe de tabela com sua fixture. Você também pode usar isso para recuperar o nome da tabela:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['model' => 'Articles'];
}
```

Como isso usa `TableRegistry::getTableLocator()->get()`, ele também suporta a sintaxe do plugin.

Naturalmente, você pode importar sua definição de tabela de um modelo/tabela existente, mas ter seus registros definidos diretamente no aparelho, como foi mostrado na seção anterior. Por exemplo:

```
class ArticlesFixture extends TestFixture
{
    public $import = ['table' => 'articles'];
    public $records = [
        [
          'title' => 'First Article',
          'body' => 'First Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:39:23',
          'modified' => '2007-03-18 10:41:31'
        ],
        [
          'title' => 'Second Article',
          'body' => 'Second Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:41:23',
          'modified' => '2007-03-18 10:43:31'
        ],
        [
          'title' => 'Third Article',
          'body' => 'Third Article Body',
          'published' => '1',
          'created' => '2007-03-18 10:43:23',
          'modified' => '2007-03-18 10:45:31'
        ]
    ];
}
```

Finalizando, não é possível carregar/criar nenhum esquema em uma fixture. Isso é útil se você já tiver uma configuração de banco de dados de teste com todas as tabelas vazias criadas. Ao não definir `$fields` nem `$import`, um equipamento

apenas inserirá seus registros e truncará os registros em cada método de teste.

Carregando Fixtures em seus Casos de Teste

Depois de criar suas fixtures, convém usá-los em seus casos de teste. Em cada caso de teste, você deve carregar as fixtures necessárias. Você deve carregar uma fixture para cada modelo que terá uma consulta executada nele. Para carregar a fixture, defina a propriedade `$fixtures` no seu modelo:

```
class ArticlesTest extends TestCase
{
    public $fixtures = ['app.Articles', 'app.Comments'];
}
```

O item acima carregará os fixtures de Article e Coment do diretório fixture do aplicativo. Você também pode carregar fixture do core do CakePHP ou plugins:

```
class ArticlesTest extends TestCase
{
    public $fixtures = [
        'plugin.DebugKit.Articles',
        'plugin.MyVendorName/MyPlugin.Messages',
        'core.Comments'
    ];
}
```

Usar o prefixo `core` carregará fixtures do CakePHP e, usando o nome de um plugin como prefixo, carregará o fixture do plugin nomeado.

Você pode controlar quando seus fixtures são carregados configurando `Cake\TestSuite\TestCase::$autoFixtures` para `false` e carregá-los posteriormente usando `Cake\TestSuite\TestCase::loadFixtures()`:

```
class ArticlesTest extends TestCase
{
    public $fixtures = ['app.Articles', 'app.Comments'];
    public $autoFixtures = false;

    public function testMyFunction()
    {
        $this->loadFixtures('Articles', 'Comments');
    }
}
```

Você pode carregar fixtures em subdiretórios. O uso de vários diretórios pode facilitar a organização de suas fixtures, se você tiver um aplicativo maior. Para carregar fixtures em subdiretórios, basta incluir o nome do subdiretório no nome do fixtures:

```
class ArticlesTest extends CakeTestCase
{
    public $fixtures = ['app.Blog/Articles', 'app.Blog/Comments'];
}
```

No exemplo acima, ambos os aparelhos seriam carregados a partir de `tests/Fixture/Blog/`.

Classes de Tabela de Teste

Digamos que já temos nossa classe de tabela de artigos definida em **src/Model/Table/ArticlesTable.php** e se parece com:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\ORM\Query;

class ArticlesTable extends Table
{
    public function findPublished(Query $query, array $options)
    {
        $query->where([
            $this->alias() . '.published' => 1
        ]);

        return $query;
    }
}
```

Agora, queremos configurar um teste que verifique esta classe de tabela. Vamos agora criar um arquivo chamado **ArticlesTableTest.php** no seu diretório **tests/TestCase/Model/Table**, com o seguinte conteúdo:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
    public $fixtures = ['app.Articles'];
}
```

Na variável de nossos casos de teste `$fixtures`, definimos o conjunto de fixtures que usaremos. Lembre-se de incluir todas as fixtures que terão consultas executadas em comparação a eles.

Criando um Método de Teste

Vamos agora adicionar um método para testar a função `publish()` na tabela Articles. Edite o arquivo **tests/TestCase/Model/Table/ArticlesTableTest.php** para que agora fique assim:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\ArticlesTable;
use Cake\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class ArticlesTableTest extends TestCase
{
```

(continues on next page)

(continuação da página anterior)

```
    public $fixtures = ['app.Articles'];

    public function setUp()
    {
        parent::setUp();
        $this->Articles = TableRegistry::getTableLocator()->get('Articles');
    }

    public function testFindPublished()
    {
        $query = $this->Articles->find('published');
        $this->assertInstanceOf('Cake\ORM\Query', $query);
        $result = $query->enableHydration(false)->toArray();
        $expected = [
            ['id' => 1, 'title' => 'First Article'],
            ['id' => 2, 'title' => 'Second Article'],
            ['id' => 3, 'title' => 'Third Article']
        ];

        $this->assertEquals($expected, $result);
    }
}
```

Você pode ver que adicionamos um método chamado `testFindPublished()`. Começamos criando uma instância da classe `ArticlesTable` e, em seguida, executamos o método `find('Published')`. Em `$expected`, definimos o que esperamos que seja o resultado adequado (que sabemos desde que definimos quais registros são preenchidos inicialmente na tabela de artigos). Testamos que o resultado é igual à nossa expectativa usando o método `assertEquals()`. Veja a seção *Executando Testes* para obter mais informações sobre como executar seu caso de teste.

Métodos Mocks de Modelo

Haverá momentos em que você desejará burlar métodos nos modelos ao testá-los. Você deve usar `getMockForModel` para criar simulações de teste de classes de tabela. Isso evita problemas com propriedades refletidas que as burlações (mocking) normais possuem:

```
public function testSendingEmails()
{
    $model = $this->getMockForModel('EmailVerification', ['send']);
    $model->expects($this->once())
        ->method('send')
        ->will($this->returnValue(true));

    $model->verifyEmail('test@example.com');
}
```

No método `tearDown()`, remova o mock com:

```
TableRegistry::clear();
```

Teste de Integração do Controlador

Embora você possa testar as classes de controladores de maneira semelhante aos Helpers, Models e Components, o CakePHP oferece uma trait especializada de nome `IntegrationTestTrait`. O uso dessa trait nos casos de teste do controlador permite realizar testes de alto nível.

Se você não está familiarizado com o teste de integração, o teste de integração é uma abordagem que facilita a verificação de várias unidades em conjunto. Os recursos de teste de integração no CakePHP simulam uma solicitação HTTP sendo tratada pelo seu aplicativo. Por exemplo, testar seu controlador também exercitará quaisquer componentes, modelos e auxiliares envolvidos no processamento de uma determinada solicitação. Isso oferece um teste de alto nível da sua aplicação e de todas as suas partes de trabalho.

Digamos que você tenha um ArticlesController típico e seu modelo correspondente. O código do controlador se parece com:

```
namespace App\Controller;

use App\Controller\AppController;

class ArticlesController extends AppController
{
    public $helpers = ['Form', 'Html'];

    public function index($short = null)
    {
        if ($this->request->is('post')) {
            $article = $this->Articles->newEntity($this->request->getData());
            if ($this->Articles->save($article)) {
                // Redirect as per PRG pattern
                return $this->redirect(['action' => 'index']);
            }
        }
        if (!empty($short)) {
            $result = $this->Articles->find('all', [
                'fields' => ['id', 'title']
            ]);
        } else {
            $result = $this->Articles->find();
        }

        $this->set([
            'title' => 'Articles',
            'articles' => $result
        ]);
    }
}
```

Crie um arquivo chamado **ArticlesControllerTest.php** em seu diretório **tests/TestCase/Controller** e coloque o seguinte dentro:

```
namespace App\Test\TestCase\Controller;

use Cake\ORM\TableRegistry;
use Cake\TestSuite\IntegrationTestTrait;
```

(continues on next page)

(continuação da página anterior)

```php
use Cake\TestSuite\TestCase;

class ArticlesControllerTest extends TestCase
{
    use IntegrationTestTrait;

    public $fixtures = ['app.Articles'];

    public function testIndex()
    {
        $this->get('/articles');

        $this->assertResponseOk();
        // Mais asserts.
    }

    public function testIndexQueryData()
    {
        $this->get('/articles?page=1');

        $this->assertResponseOk();
        // Mais asserts.
    }

    public function testIndexShort()
    {
        $this->get('/articles/index/short');

        $this->assertResponseOk();
        $this->assertResponseContains('Articles');
        // Mais asserts.
    }

    public function testIndexPostData()
    {
        $data = [
            'user_id' => 1,
            'published' => 1,
            'slug' => 'new-article',
            'title' => 'New Article',
            'body' => 'New Body'
        ];
        $this->post('/articles', $data);

        $this->assertResponseSuccess();
        $articles = TableRegistry::getTableLocator()->get('Articles');
        $query = $articles->find()->where(['title' => $data['title']]);
        $this->assertEquals(1, $query->count());
    }
}
```

Este exemplo mostra alguns dos métodos de envio de solicitação e algumas das asserções que o `IntegrationTestTrait` fornece. Antes de fazer qualquer afirmação, você precisará enviar uma solicitação.

Você pode usar um dos seguintes métodos para enviar uma solicitação:

- `get()` Envia uma solicitação GET.
- `post()` Envia uma solicitação POST.
- `put()` Envia uma solicitação PUT.
- `delete()` Envia uma solicitação DELETE.
- `patch()` Envia uma solicitação PATCH.
- `options()` Envia uma solicitação OPTIONS.
- `head()` Envia uma solicitação HEAD.

Todos os métodos, exceto `get()` e `delete()`, aceitam um segundo parâmetro que permite enviar um corpo de solicitação. Depois de enviar uma solicitação, você pode usar as várias asserções fornecidas por `IntegrationTestTrait` ou PHPUnit para garantir que sua solicitação tenha os efeitos colaterais corretos.

Configurando a Solicitação

A trait `IntegrationTestTrait` vem com vários métodos auxiliares para facilitar a configuração das solicitações que você enviará ao seu aplicativo em teste:

```
// Configura cookies
$this->cookie('name', 'Uncle Bob');

// Defina um valor na sessão
$this->session(['Auth.User.id' => 1]);

// Configura cabeçalhos
$this->configRequest([
    'headers' => ['Accept' => 'application/json']
]);
```

O estado definido por esses métodos auxiliares é redefinido no método `tearDown()`.

Testando Ações que Exigem Autenticação

Se você estiver usando `AuthComponent`, precisará remover os dados da sessão que o AuthComponent usa para validar a identidade de um usuário. Você pode usar métodos auxiliares em `IntegrationTestTrait` para fazer isso. Supondo que você tivesse um `ArticlesController` que continha um método add e que exigisse autenticação com o método add, você poderia escrever os seguintes testes:

```
public function testAddUnauthenticatedFails()
{
    // Nenhum conjunto de dados da sessão.
    $this->get('/articles/add');

    $this->assertRedirect(['controller' => 'Users', 'action' => 'login']);
}

public function testAddAuthenticated()
{
    // Define dados da sessão
```

(continues on next page)

(continuação da página anterior)

```
    $this->session([
        'Auth' => [
            'User' => [
                'id' => 1,
                'username' => 'testing',
                // outras chaves
            ]
        ]
    ]);
    $this->get('/articles/add');

    $this->assertResponseOk();
    // Outras asserts.
}
```

Testando Autenticação Stateless e APIs

Para testar APIs que usam autenticação sem estado, como autenticação Básica, você pode configurar a solicitação para injetar condições do ambiente ou cabeçalhos que simulam cabeçalhos de solicitação de autenticação reais.

Ao testar a autenticação Básica ou Digest, você pode adicionar as variáveis de ambiente que o *PHP cria <https://php.net/manual/en/features.http-auth.php>* automaticamente. Essas variáveis de ambiente usadas no adaptador de autenticação descritas em *Usando Autenticação Básica*:

```
public function testBasicAuthentication()
{
    $this->configRequest([
        'environment' => [
            'PHP_AUTH_USER' => 'username',
            'PHP_AUTH_PW' => 'password',
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

Se você estiver testando outras formas de autenticação, como OAuth2, poderá definir o cabeçalho de Autorização diretamente:

```
public function testOauthToken()
{
    $this->configRequest([
        'headers' => [
            'authorization' => 'Bearer: oauth-token'
        ]
    ]);

    $this->get('/api/posts');
    $this->assertResponseOk();
}
```

A chave de cabeçalhos em `configRequest()` pode ser usada para configurar qualquer cabeçalho HTTP adicional necessário para uma ação.

Testando Ações Protegidas por CsrfComponent ou SecurityComponent

Ao testar ações protegidas por SecurityComponent ou CsrfComponent, você pode ativar a geração automática de token para garantir que seus testes não falhem devido a incompatibilidades de token:

```
public function testAdd()
{
    $this->enableCsrfToken();
    $this->enableSecurityToken();
    $this->post('/posts/add', ['title' => 'Exciting news!']);
}
```

Também é importante habilitar a depuração em testes que usam tokens para impedir que o SecurityComponent pense que o token de depuração está sendo usado em um ambiente sem depuração. Ao testar com outros métodos como `requireSecure()`, você pode usar `configRequest()` para definir as variáveis de ambiente corretas:

```
// Falsificar conexões SSL.
$this->configRequest([
    'environment' => ['HTTPS' => 'on']
]);
```

Teste de Integração PSR-7 Middleware

O teste de integração também pode ser usado para testar todo o aplicativo PSR-7 e *Middleware*. Por padrão, o `IntegrationTestTrait` detecta automaticamente a presença de uma classe `App\Application` e habilita automaticamente o teste de integração do seu aplicativo. Você pode alternar esse comportamento com o método `useHttpServer()`:

```
public function setUp()
{
    // Ative o teste de integração PSR-7.
    $this->useHttpServer(true);

    // Desative o teste de integração PSR-7.
    $this->useHttpServer(false);
}
```

Você pode personalizar o nome da classe do aplicativo usado e os argumentos do construtor, usando o método `configApplication()`:

```
public function setUp()
{
    $this->configApplication('App\App', [CONFIG]);
}
```

Depois de ativar o modo PSR-7 e, possivelmente, configurar sua classe de aplicativo, você pode usar os recursos restantes do `IntegrationTestTrait` normalmente.

Você também deve tentar usar *Application::bootstrap()* para carregar qualquer plug-in que contenha eventos/rotas. Isso garantirá que seus eventos/rotas estejam conectados para cada caso de teste. Como alternativa, se você deseja carregar

plug-ins manualmente em um teste, pode usar o método `loadPlugins()`.

Testando com Cookies Criptografados

Se você usar `Cake\Controller\Component\CookieComponent` em seus controladores, é provável que seus cookies sejam criptografados. A partir do 3.1.7, o CakePHP fornece métodos auxiliares para interagir com cookies criptografados nos seus casos de teste:

```
// Defina um cookie usando o AES e a chave padrão.
$this->cookieEncrypted('my_cookie', 'Some secret values');

// Suponha que esta ação modifique o cookie.
$this->get('/bookmarks/index');

$this->assertCookieEncrypted('An updated value', 'my_cookie');
```

Testando Mensagens Flash

Se você deseja testar a presença de mensagens flash na sessão e não o HTML renderizado, pode usar `enableRetainFlashMessages()` em seus testes para reter mensagens flash na sessão, para poder escrever as assertions:

```
$this->enableRetainFlashMessages();
$this->get('/bookmarks/delete/9999');

$this->assertSession('That bookmark does not exist', 'Flash.flash.0.message');
```

A partir da versão 3.7.0, existem auxiliares de teste adicionais para mensagens flash:

```
$this->enableRetainFlashMessages();
$this->get('/bookmarks/delete/9999');

// Coloque uma mensagem flash na chave 'flash'.
$this->assertFlashMessage('Bookmark deleted', 'flash');

// Afirme a segunda mensagem flash, também na chave 'flash'.
$this->assertFlashMessageAt(1, 'Bookmark really deleted');

// Afirme uma mensagem flash na chave 'auth' na primeira posição
$this->assertFlashMessageAt(0, 'You are not allowed to enter this dungeon!', 'auth');

// Afirmar que uma mensagem flash usa o elemento error
$this->assertFlashElement('Flash/error');

// Afirme o segundo elemento de mensagem flash
$this->assertFlashElementAt(1, 'Flash/error');
```

Testando um Controlador Com Resposta em JSON

JSON é um formato amigável e comum a ser usado ao criar um serviço da web. Testar os pontos finais do seu serviço da web é muito simples com o CakePHP. Vamos começar com um exemplo simples de controlador que responde em JSON:

```
class MarkersController extends AppController
{
    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('RequestHandler');
    }

    public function view($id)
    {
        $marker = $this->Markers->get($id);
        $this->set([
            '_serialize' => ['marker'],
            'marker' => $marker,
        ]);
    }
}
```

Agora, criamos o arquivo **tests/TestCase/Controller/MarkersControllerTest.php** e garantimos que nosso serviço da Web retorne a resposta adequada:

```
class MarkersControllerTest extends IntegrationTestCase
{
    public function testGet()
    {
        $this->configRequest([
            'headers' => ['Accept' => 'application/json']
        ]);
        $result = $this->get('/markers/view/1.json');

        // Check that the response was a 200
        $this->assertResponseOk();

        $expected = [
            ['id' => 1, 'lng' => 66, 'lat' => 45],
        ];
        $expected = json_encode($expected, JSON_PRETTY_PRINT);
        $this->assertEquals($expected, (string)$this->_response->getBody());
    }
}
```

Nós usamos a opção `JSON_PRETTY_PRINT`, pois o CakePHP embutido no JsonView usará essa opção quando `debug` estiver ativado.

Teste com carregamentos de ficheiros

A simulação de carregamentos de ficheiros é simples quando se utiliza o modo padrão *arquivos carregados como objectos <request-file-uploads>*. Pode simplesmente criar instâncias que implementem \Psr\Http\Message\UploadedFileInterface[129] (a implementação padrão actualmente utilizada pelo CakePHP é `\Laminas\Diactoros\UploadedFile`), e passá-los nos seus dados de pedido de teste. No ambiente CLI, tais objectos irão, por defeito, passar na validação que testa se o ficheiro foi carregado via HTTP. O mesmo não é verdade para os dados de estilo array como os encontrados em `$_FILES`, falharia essa verificação.

A fim de simular exactamente como os objectos de ficheiro carregados estariam presentes num pedido regular, é necessário não só passá-los nos dados do pedido, mas também passá-los para a configuração do pedido de teste através da opção `files`. Mas não é tecnicamente necessário, a menos que o seu código aceda aos ficheiros carregados através dos métodos `Cake\Http\ServerRequest::getUploadedFile()` ou `Cake\Http\ServerRequest::getUploadedFiles()`.

Vamos assumir que os artigos têm uma imagem teaser, e uma associação `Articles hasMany Attachments`, o formulário pareceria algo parecido com isto em conformidade, onde um ficheiro de imagem, e múltiplos anexos/arquivos seriam aceites:

```
<?= $this->Form->create($article, ['type' => 'file']) ?>
<?= $this->Form->control('title') ?>
<?= $this->Form->control('teaser_image', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.0.description']) ?>
<?= $this->Form->control('attachments.1.attachment', ['type' => 'file']) ?>
<?= $this->Form->control('attachments.1.description']) ?>
<?= $this->Form->button('Submit') ?>
<?= $this->Form->end() ?>
```

O teste que simularia o pedido correspondente poderia parecer-se com o seguinte:

```
public function testAddWithUploads(): void
{
    $teaserImage = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.jpg', // fluxo ou caminho para o ficheiro que representa o␣
↪ficheiro temporário
        12345,                    // os ficheiros em bytes
        \UPLOAD_ERR_OK,           // o estado de carregamento/erro
        'teaser.jpg',             // o nome do ficheiro tal como enviado pelo cliente
        'image/jpeg'              // a mimetype tal como enviada pelo cliente
    );

    $textAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.txt',
        'text/plain'
    );

    $pdfAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.pdf',
        12345,
```

(continues on next page)

[129] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

(continuação da página anterior)

```
        \UPLOAD_ERR_OK,
        'attachment.pdf',
        'application/pdf'
    );

    // Estes são os dados acessíveis através de `$this->request->getUploadedFile()`
    // e `$this->request->getUploadedFiles()`.
    $this->configRequest([
        'files' => [
            'teaser_image' => $teaserImage,
            'attachments' => [
                0 => [
                    'attachment' => $textAttachment,
                ],
                1 => [
                    'attachment' => $pdfAttachment,
                ],
            ],
        ],
    ]);

    // Estes são os dados acessíveis através de  `$this->request->getData()`.
    $postData = [
        'title' => 'Novo Artigo',
        'teaser_image' => $teaserImage,
        'attachments' => [
            0 => [
                'attachment' => $textAttachment,
                'description' => 'Text attachment',
            ],
            1 => [
                'attachment' => $pdfAttachment,
                'description' => 'PDF attachment',
            ],
        ],
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage('O artigo foi salvo com sucesso');
    $this->assertFileExists('/path/to/uploads/teaser.jpg');
    $this->assertFileExists('/path/to/uploads/attachment.txt');
    $this->assertFileExists('/path/to/uploads/attachment.pdf');
}
```

Dica: Se configurar o pedido de teste com ficheiros, então ele *terá* de corresponder à estrutura dos seus dados POST (mas apenas incluir os objectos de ficheiro carregados)!

Da mesma forma, pode simular erros de carregamento[130] ou ficheiros inválidos que não passem na validação:

[130] https://www.php.net/manual/en/features.file-upload.errors.php

```
public function testAddWithInvalidUploads(): void
{
    $missingTeaserImageUpload = new \Laminas\Diactoros\UploadedFile(
        '',
        0,
        \UPLOAD_ERR_NO_FILE,
        '',
        ''
    );

    $uploadFailureAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.txt',
        1234567890,
        \UPLOAD_ERR_INI_SIZE,
        'attachment.txt',
        'text/plain'
    );

    $invalidTypeAttachment = new \Laminas\Diactoros\UploadedFile(
        '/path/to/test/file.exe',
        12345,
        \UPLOAD_ERR_OK,
        'attachment.exe',
        'application/vnd.microsoft.portable-executable'
    );

    $this->configRequest([
        'files' => [
            'teaser_image' => $missingTeaserImageUpload,
            'attachments' => [
                0 => [
                    'file' => $uploadFailureAttachment,
                ],
                1 => [
                    'file' => $invalidTypeAttachment,
                ],
            ],
        ],
    ]);

    $postData = [
        'title' => 'Novo Artigo',
        'teaser_image' => $missingTeaserImageUpload,
        'attachments' => [
            0 => [
                'file' => $uploadFailureAttachment,
                'description' => 'Upload de anexo de falha',
            ],
            1 => [
                'file' => $invalidTypeAttachment,
                'description' => 'Fixação de tipo inválido',
            ],
        ],
```

(continues on next page)

(continuação da página anterior)

```
    ];
    $this->post('/articles/add', $postData);

    $this->assertResponseOk();
    $this->assertFlashMessage('O artigo não pôde ser salvo');
    $this->assertResponseContains('É necessária uma imagem de teaser');
    $this->assertResponseContains('Tamanho máximo de ficheiros permitido excedido');
    $this->assertResponseContains('Tipo de ficheiro não suportado');
    $this->assertFileNotExists('/path/to/uploads/teaser.jpg');
    $this->assertFileNotExists('/path/to/uploads/attachment.txt');
    $this->assertFileNotExists('/path/to/uploads/attachment.exe');
}
```

Desabilitando o Tratamento de Erros de Middlewares nos Testes

Ao depurar testes que estão falhando porque seu aplicativo está encontrando erros, pode ser útil desativar temporariamente o middleware de manipulação de erros para permitir que o erro subjacente seja exibido. Você pode usar o método `disableErrorHandlerMiddleware()` para fazer isso:

```
public function testGetMissing()
{
    $this->disableErrorHandlerMiddleware();
    $this->get('/markers/not-there');
    $this->assertResponseCode(404);
}
```

No exemplo acima, o teste falharia e a mensagem de exceção subjacente e o rastreamento da pilha seriam exibidos em vez da verificação da página de erro renderizada.

Métodos Assertion

A característica `IntegrationTestTrait` fornece vários métodos de asserção que tornam as respostas de teste muito mais simples. Alguns exemplos são:

```
// Verifica se o código da resposta é 2xx
$this->assertResponseOk();

// Verifica se o código de resposta é 2xx/3xx
$this->assertResponseSuccess();

// Verifica se o código de resposta é 4xx
$this->assertResponseError();

// Verifica se o código de resposta 5xx
$this->assertResponseFailure();

// Verifica se a resposta tem um código específico, exemplo: 200
$this->assertResponseCode(200);

// Verifica o cabeçalho do local
```

(continues on next page)

(continuação da página anterior)

```
$this->assertRedirect(['controller' => 'Articles', 'action' => 'index']);

// Verifica se nenhum cabeçalho de redirecionamento foi definido
$this->assertNoRedirect();

// Verifique uma parte do cabeçalho Location
$this->assertRedirectContains('/articles/edit/');

// Adicionado em 3.7.0
$this->assertRedirectNotContains('/articles/edit/');

// Verifica se conteúdo de resposta não está vazio
$this->assertResponseNotEmpty();

// Verifica conteúdo de resposta vazio
$this->assertResponseEmpty();

// Afirmar conteúdo de resposta
$this->assertResponseEquals('Yeah!');

// Afirmar que o conteúdo da resposta não é igual ao especifícado
$this->assertResponseNotEquals('No!');

// Afirmar conteúdo de resposta parcialmente
$this->assertResponseContains('You won!');
$this->assertResponseNotContains('You lost!');

// Afirmar arquivo enviado de volta
$this->assertFileResponse('/absolute/path/to/file.ext');

// Afirmar layout
$this->assertLayout('default');

// Afirme qual modelo foi renderizado (se houver)
$this->assertTemplate('index');

// Afirmar dados na sessão
$this->assertSession(1, 'Auth.User.id');

// Afirmar cabeçalho de resposta.
$this->assertHeader('Content-Type', 'application/json');
$this->assertHeaderContains('Content-Type', 'html');

// Adicionado em 3.7.0
$this->assertHeaderNotContains('Content-Type', 'xml');

// Afirmar variáveis de exibição
$user =  $this->viewVariable('user');
$this->assertEquals('jose', $user->username);

// Afirmar cookies na resposta
$this->assertCookie('1', 'thingid');
```

(continues on next page)

(continuação da página anterior)

```
// Verifique o tipo de conteúdo
$this->assertContentType('application/json');
```

Além dos métodos de asserção acima, você também pode usar todas as asserções no TestSuite[131] e os encontrados em PHPUnit[132].

Comparando Resultados de Teste com um Arquivo

Para alguns tipos de teste, pode ser mais fácil comparar o resultado de um teste com o conteúdo de um arquivo - por exemplo, ao testar a saída renderizada de uma visualização. O `StringCompareTrait` adiciona um método de declaração simples para essa finalidade.

O uso envolve o uso da característica, definindo o caminho base de comparação e chamando `assertSameAsFile`:

```
use Cake\TestSuite\StringCompareTrait;
use Cake\TestSuite\TestCase;

class SomeTest extends TestCase
{
    use StringCompareTrait;

    public function setUp()
    {
        $this->_compareBasePath = APP . 'tests' . DS . 'comparisons' . DS;
        parent::setUp();
    }

    public function testExample()
    {
        $result = ...;
        $this->assertSameAsFile('example.php', $result);
    }
}
```

O exemplo acima comparará `$result` com o conteúdo do arquivo `APP/tests/comparisons/example.php`.

Um mecanismo é fornecido para gravar/atualizar arquivos de teste, configurando a variável de ambiente `UPDATE_TEST_COMPARISON_FILES`, que criará e/ou atualizará os arquivos de comparação de testes à medida que forem referenciados:

```
phpunit
...
FAILURES!
Tests: 6, Assertions: 7, Failures: 1

UPDATE_TEST_COMPARISON_FILES=1 phpunit
...
OK (6 tests, 7 assertions)
```

(continues on next page)

[131] https://api.cakephp.org/4.x/class-Cake.TestSuite.TestCase.html
[132] https://phpunit.de/manual/current/en/appendixes.assertions.html

(continuação da página anterior)

```
git status
...
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   tests/comparisons/example.php
```

Teste de Integração de Console

Veja *console-integration-testing* para obter informações sobre testes de shells e comandos.

Testando Views

Geralmente a maioria dos aplicativos não testa diretamente seu código HTML. Fazer isso geralmente resulta em conjuntos de testes frágeis e difíceis de manter, com tendência a serem quebrados. Ao escrever testes funcionais usando `IntegrationTestTrait`, você pode inspecionar o conteúdo da visualização renderizada configurando a opção` *return* `para "view". Embora seja possível testar o conteúdo da visualização usando `IntegrationTestTrait`, um teste de integração/visualização mais robusto e sustentável pode ser realizado usando ferramentas como Selenium webdriver[133].

Testando Componentes

Vamos fingir que temos um componente chamado PagematronComponent em nosso aplicativo. Esse componente nos ajuda a definir o valor limite de paginação em todos os controladores que o utilizam. Aqui está o nosso exemplo de componente localizado em **src/Controller/Component/PagematronComponent.php**:

```
class PagematronComponent extends Component
{
    public $controller = null;

    public function setController($controller)
    {
        $this->controller = $controller;
        // Verifique se o controlador está usando paginação
        if (!isset($this->controller->paginate)) {
            $this->controller->paginate = [];
        }
    }

    public function startup(Event $event)
    {
        $this->setController($event->getSubject());
    }
```

(continues on next page)

[133] https://www.selenium.dev/

(continuação da página anterior)

```
    public function adjust($length = 'short')
    {
        switch ($length) {
            case 'long':
                $this->controller->paginate['limit'] = 100;
            break;
            case 'medium':
                $this->controller->paginate['limit'] = 50;
            break;
            default:
                $this->controller->paginate['limit'] = 20;
            break;
        }
    }
}
```

Agora podemos escrever testes para garantir que nosso parâmetro paginado `limit` esteja sendo definido corretamente pelo método `Adjust()` em nosso componente. Criamos o arquivo **tests/TestCase/Controller/Component/PagematronComponentTest.php**:

```
namespace App\Test\TestCase\Controller\Component;

use App\Controller\Component\PagematronComponent;
use Cake\Controller\Controller;
use Cake\Controller\ComponentRegistry;
use Cake\Event\Event;
use Cake\Http\ServerRequest;
use Cake\Http\Response;
use Cake\TestSuite\TestCase;

class PagematronComponentTest extends TestCase
{

    public $component = null;
    public $controller = null;

    public function setUp()
    {
        parent::setUp();
        // Configure nosso componente e o controlador de teste fake
        $request = new ServerRequest();
        $response = new Response();
        $this->controller = $this->getMockBuilder('Cake\Controller\Controller')
            ->setConstructorArgs([$request, $response])
            ->setMethods(null)
            ->getMock();
        $registry = new ComponentRegistry($this->controller);
        $this->component = new PagematronComponent($registry);
        $event = new Event('Controller.startup', $this->controller);
        $this->component->startup($event);
    }
```

(continues on next page)

(continuação da página anterior)

```
    public function testAdjust()
    {
        // Teste nosso método de ajuste com diferentes configurações de parâmetros
        $this->component->adjust();
        $this->assertEquals(20, $this->controller->paginate['limit']);

        $this->component->adjust('medium');
        $this->assertEquals(50, $this->controller->paginate['limit']);

        $this->component->adjust('long');
        $this->assertEquals(100, $this->controller->paginate['limit']);
    }

    public function tearDown()
    {
        parent::tearDown();
        // Limpar depois que terminarmos
        unset($this->component, $this->controller);
    }
}
```

Testando Ajudantes

Como uma quantidade decente de lógica reside nas classes Helper, é importante garantir que essas classes sejam cobertas por casos de teste.

Primeiro, criamos um exemplo de auxiliar para testar. O `CurrencyRendererHelper` nos ajudará a exibir moedas em nossos pontos de vista e, por simplicidade, só possui um método `usd()`:

```
// src/View/Helper/CurrencyRendererHelper.php
namespace App\View\Helper;

use Cake\View\Helper;

class CurrencyRendererHelper extends Helper
{
    public function usd($amount)
    {
        return 'USD ' . number_format($amount, 2, '.', ',');
    }
}
```

Aqui, definimos as casas decimais como 2, separador decimal para ponto, separador de milhares para vírgula e prefixamos o número formatado com a string “USD”.

Agora criamos nossos testes:

```
// tests/TestCase/View/Helper/CurrencyRendererHelperTest.php

namespace App\Test\TestCase\View\Helper;
```

(continues on next page)

(continuação da página anterior)

```
use App\View\Helper\CurrencyRendererHelper;
use Cake\TestSuite\TestCase;
use Cake\View\View;

class CurrencyRendererHelperTest extends TestCase
{
    public $helper = null;

    // Aqui instanciamos nosso ajudante
    public function setUp()
    {
        parent::setUp();
        $View = new View();
        $this->helper = new CurrencyRendererHelper($View);
    }

    // Testando a função usd()
    public function testUsd()
    {
        $this->assertEquals('USD 5.30', $this->helper->usd(5.30));

        // Devemos sempre ter 2 dígitos decimais
        $this->assertEquals('USD 1.00', $this->helper->usd(1));
        $this->assertEquals('USD 2.05', $this->helper->usd(2.05));

        // Testando o separador de milhares
        $this->assertEquals(
          'USD 12,000.70',
          $this->helper->usd(12000.70)
        );
    }
}
```

Aqui, chamamos `usd()` com parâmetros diferentes e dizemos ao conjunto de testes para verificar se os valores retornados são iguais ao esperado.

Salve isso e execute o teste. Você deverá ver uma barra verde e mensagens indicando 1 passe e 4 asserções.

Quando você estiver testando um Helper que use outros helpers, «mock» o método `loadHelpers` da classe View.

Testando Eventos

O *Sistema de Eventos* é uma ótima maneira de desacoplar o código do aplicativo, mas às vezes ao testar, você tende a testar os resultados dos eventos nos casos de teste que os executam. Esta é uma forma adicional de acoplamento que pode ser removida usando `assertEventFired` e `assertEventFiredWith`.

Expandindo no exemplo Orders, digamos que temos as seguintes tabelas:

```
class OrdersTable extends Table
{
    public function place($order)
    {
```

(continues on next page)

(continuação da página anterior)

```
        if ($this->save($order)) {
            // remoção de carrinho movido para CartsTable
            $event = new Event('Model.Order.afterPlace', $this, [
                'order' => $order
            ]);
            $this->eventManager()->dispatch($event);

            return true;
        }

        return false;
    }
}

class CartsTable extends Table
{
    public function implementedEvents()
    {
        return [
            'Model.Order.afterPlace' => 'removeFromCart'
        ];
    }

    public function removeFromCart(Event $event)
    {
        $order = $event->getData('order');
        $this->delete($order->cart_id);
    }
}
```

Nota: Para afirmar que os eventos foram disparados, você deve primeiro ativar *Rastreando Eventos* no gerenciador de eventos que deseja reivindicar.

Para testar o `OrdersTable` acima, habilitamos o rastreamento em `setUp()`, depois afirmamos que o evento foi disparado e afirmamos que a entidade `$order` foi passada nos dados do evento:

```
namespace App\Test\TestCase\Model\Table;

use App\Model\Table\OrdersTable;
use Cake\Event\EventList;
use Cake\ORM\TableRegistry;
use Cake\TestSuite\TestCase;

class OrdersTableTest extends TestCase
{
    public $fixtures = ['app.Orders'];

    public function setUp()
    {
        parent::setUp();
        $this->Orders = TableRegistry::getTableLocator()->get('Orders');
```

(continues on next page)

(continuação da página anterior)

```
        // ativar o rastreamento de eventos
        $this->Orders->getEventManager()->setEventList(new EventList());
    }

    public function testPlace()
    {
        $order = new Order([
            'user_id' => 1,
            'item' => 'Cake',
            'quantity' => 42,
        ]);

        $this->assertTrue($this->Orders->place($order));

        $this->assertEventFired('Model.Order.afterPlace', $this->Orders->
↪getEventManager());
        $this->assertEventFiredWith('Model.Order.afterPlace', 'order', $order, $this->
↪Orders->getEventManager());
    }
}
```

Por padrão, o global `EventManager` é usado para asserções, portanto, testar eventos globais não requer a aprovação do gerenciador de eventos:

```
$this->assertEventFired('My.Global.Event');
$this->assertEventFiredWith('My.Global.Event', 'user', 1);
```

Testando Email

Veja *Testando Emails* para obter informações sobre o teste de email.

Criando Suítes de Teste

Se você deseja que vários de seus testes sejam executados ao mesmo tempo, é possível criar um conjunto de testes. Um conjunto de testes é composto por vários casos de teste. Você pode criar suítes de teste no arquivo **phpunit.xml** do seu aplicativo. Um exemplo simples seria:

```
<testsuites>
  <testsuite name="Models">
    <directory>src/Model</directory>
    <file>src/Service/UserServiceTest.php</file>
    <exclude>src/Model/Cloud/ImagesTest.php</exclude>
  </testsuite>
</testsuites>
```

Criando Testes para Plugins

Os testes para plugins são criados em seu próprio diretório, dentro da pasta plugins.:

```
/src
/plugins
    /Blog
        /tests
            /TestCase
            /Fixture
```

Eles funcionam como testes normais, mas você deve se lembrar de usar as convenções de nomenclatura para plug-ins ao importar classes. Este é um exemplo de uma caixa de teste para o modelo `BlogPost` do capítulo de plugins deste manual. A diferença de outros testes está na primeira linha em que "Blog.BlogPost" é importado. Você também precisa prefixar os dispositivos de seu plugin com `plugin.Blog.BlogPosts`:

```
namespace Blog\Test\TestCase\Model\Table;

use Blog\Model\Table\BlogPostsTable;
use Cake\TestSuite\TestCase;

class BlogPostsTableTest extends TestCase
{
    // Acessórios para plug-ins localizados em /plugins/Blog/tests/Fixture/
    public $fixtures = ['plugin.Blog.BlogPosts'];

    public function testSomething()
    {
        // Teste alguma coisa.
    }
}
```

Se você deseja usar fixtures de plug-in nos testes do aplicativo, pode referenciá-los usando a sintaxe `plugin.pluginName.fixtureName` na matriz `$fixtures`. Além disso, se você usar o nome do plugin do fornecedor ou os diretórios do equipamento, poderá usar o seguinte: `plugin.vendorName/pluginName.folderName/fixtureName`.

Antes de usar os equipamentos, verifique novamente se o seu `phpunit.xml` contém o ouvinte do equipamento:

```
<!-- Configurar um ouvinte para fixtures -->
<listeners>
    <listener
    class="\Cake\TestSuite\Fixture\FixtureInjector">
        <arguments>
            <object class="\Cake\TestSuite\Fixture\FixtureManager" />
        </arguments>
    </listener>
</listeners>
```

Você também deve garantir que suas fixtures sejam carregáveis. Verifique se o seguinte arquivo está presente em seu arquivo **composer.json**:

```
"autoload-dev": {
    "psr-4": {
        "MyPlugin\\Test\\": "plugins/MyPlugin/tests/"
```

(continues on next page)

(continuação da página anterior)

```
    }
}
```

Nota: Lembre-se de executar o `composer.phar dumpautoload` ao adicionar novos mapeamentos de carregamento automático.

Gerando Testes com o Bake

Se você usar *bake* para gerar scaffolding, ele também gerará stubs de teste. Se você precisar gerar novamente esqueletos de casos de teste ou se desejar gerar esqueletos de teste para o código que escreveu, poderá usar o `bake`:

```
bin/cake bake test <type> <name>
```

`<type>` deve ser um dos:

1. Entity
2. Table
3. Controller
4. Component
5. Behavior
6. Helper
7. Shell
8. Task
9. ShellHelper
10. Cell
11. Form
12. Mailer
13. Command

`<name>` deve ser o nome do objeto para o qual você deseja criar um esqueleto de teste.

Integração com Jenkins

O Jenkins[134] é um servidor de integração contínua, que pode ajudá-lo a automatizar a execução dos seus casos de teste. Isso ajuda a garantir que todos os seus testes permaneçam aprovados e seu aplicativo esteja sempre pronto.

A integração de um aplicativo CakePHP com o Jenkins é bastante direta. O seguinte pressupõe que você já instalou o Jenkins no sistema *nix e pode administrá-lo. Você também sabe como criar jobs e executar builds. Se você não tiver certeza disso, consulte a *documentação de Jenkins <https://jenkins-ci.org/>*.

[134] https://jenkins-ci.org

Criando um Trabalho

Comece criando um trabalho para seu aplicativo e conecte seu repositório para que jenkins possa acessar seu código.

Adicionar Configuração do Banco de Dados de Teste

Usar um banco de dados separado apenas para Jenkins geralmente é uma boa idéia, pois evita vários problemas básicos. Depois de criar um novo banco de dados em um servidor de banco de dados que jenkins pode acessar (geralmente localhost). Adicione um *shell script* à compilação que contém o seguinte:

```
cat > config/app_local.php <<'CONFIG'
<?php
return [
    'Datasources' => [
        'test' => [
            'datasource' => 'Database/Mysql',
            'host'       => 'localhost',
            'database'   => 'jenkins_test',
            'username'      => 'jenkins',
            'password'   => 'cakephp_jenkins',
            'encoding'   => 'utf8'
        ]
    ]
];
CONFIG
```

Descomente a seguinte linha no seu arquivo **config/bootstrap.php**:

```
//Configure::load('app_local', 'default');
```

Ao criar um arquivo **app_local.php**, você tem uma maneira fácil de definir configurações específicas do Jenkins. Você pode usar esse mesmo arquivo de configuração para substituir qualquer outro arquivo de configuração necessário no Jenkins.

Geralmente, é uma boa ideia eliminar e recriar o banco de dados antes de cada compilação também. Isso o isola de falhas encadeadas, onde uma construção quebrada faz com que outras falhem. Adicione outra etapa do *shell script* à compilação que contém o seguinte:

```
mysql -u jenkins -pcakephp_jenkins -e 'DROP DATABASE IF EXISTS jenkins_test; CREATE␣
↪DATABASE jenkins_test';
```

Adicione seus Testes

Adicione outra etapa do *shell script* à sua compilação. Nesta etapa, instale suas dependências e execute os testes para seu aplicativo. Criar um arquivo de log junit ou cobertura de código geralmente é um bom bônus, pois fornece uma boa visualização gráfica dos resultados dos testes:

```
# Faça o download do Composer, se estiver faltando.
test -f 'composer.phar' || curl -sS https://getcomposer.org/installer | php
# Instale dependências
php composer.phar install
vendor/bin/phpunit --log-junit junit.xml --coverage-clover clover.xml
```

Se você usar a cobertura de código ou os resultados do JUnit, certifique-se de configurar também o Jenkins. Não configurar essas etapas significa que você não verá os resultados.

Executando uma Build

Agora você deve poder executar uma compilação. Verifique a saída do console e faça as alterações necessárias para obter uma compilação de aprovação.

CAPÍTULO 29

Validação

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[135] 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.

[135] https://github.com/cakephp/docs

CAPÍTULO 30

Classe App

class Cake\Core**App**

A classe App é responsável pela localização dos recursos e pelo gerenciamento de caminhos.

Encontrando Classes

static Cake\Core\App::**classname**(*$name*, *$type = ''*, *$suffix = ''*)

Este método é usado para resolver nomes de classes no CakePHP. Resolve os nomes abreviados do CakePHP e retorna o nome da classe totalmente resolvido:

```
// Resolva um nome de classe curto com o namespace + sufixo.
App::classname('Auth', 'Controller/Component', 'Component');
// Retorna Cake\Controller\Component\AuthComponent

// Resolve o nome do plugin
App::classname('DebugKit.Toolbar', 'Controller/Component', 'Component');
// Retorna DebugKit\Controller\Component\ToolbarComponent

// Os nomes com \ serão retornados inalterados.
App::classname('App\Cache\ComboCache');
// Retorna App\Cache\ComboCache
```

Ao resolver as classes, o namespace `App` será tentado e, se a classe não existir, o espaço para nome `Cake` será tentado. Se ambos os nomes de classe não existirem, `false` será retornado.

Localizando Caminhos para Namespaces

static Cake\Core\App::**path**(*string $package*, *string $plugin = null*)

Usado para obter locais para caminhos com base em convenções:

```
// Obtenha o caminho para o Controller / no seu aplicativo
App::path('Controller');
```

Isso pode ser feito para todos os namespaces que fazem parte do seu aplicativo. Você também pode buscar caminhos para um plug-in:

```
// Retorna os caminhos do componente no DebugKit
App::path('Component', 'DebugKit');
```

`App::path()` retornará apenas o caminho padrão e não poderá fornecer informações sobre caminhos adicionais para os quais o carregador automático está configurado.

static Cake\Core\App::**core**(*string $package*)

Usado para encontrar o caminho para um pacote dentro do CakePHP:

```
// Obtenha o caminho para os mecanismos de cache.
App::core('Cache/Engine');
```

Localizando Plugins

static Cake\Core\Plugin::**path**(*string $plugin*)

Os plug-ins podem ser localizados com o Plugin. Usar `Plugin::path('DebugKit');` por exemplo, fornecerá o caminho completo para o plug-in DebugKit:

```
$path = Plugin::path('DebugKit');
```

Localizando Temas

Como os temas são plugins, você pode usar os métodos acima para obter o caminho para um tema.

Carregando Arquivos do Fornecedor

O ideal é que os arquivos do fornecedor sejam carregados automaticamente com o `Composer`, se você tiver arquivos que não possam ser carregados ou instalados automaticamente com o Composer, será necessário usar o `require` para carregá-los.

Se você não conseguir instalar uma biblioteca com o Composer, é melhor instalar cada biblioteca em um diretório, seguindo a convenção do Composer de `vendor/$author/$ package`. Se você tiver uma biblioteca chamada AcmeLib, poderá instalá-la em `vendor/Acme/AcmeLib`. Supondo que ele não usasse nomes de classe compatíveis com PSR-0, você poderia carregar automaticamente as classes dentro dele usando `classmap` no `composer.json` do seu aplicativo:

```
"autoload": {
    "psr-4": {
        "App\\": "src/",
        "App\\Test\\": "tests/"
    },
    "classmap": [
        "vendor/Acme/AcmeLib"
    ]
}
```

Se a sua biblioteca de fornecedores não usa classes e, em vez disso, fornece funções, você pode configurar o Composer para carregar esses arquivos no início de cada solicitação usando a estratégia de carregamento automático `files`:

```
"autoload": {
    "psr-4": {
        "App\\": "src/",
        "App\\Test\\": "tests/"
    },
    "files": [
        "vendor/Acme/AcmeLib/functions.php"
    ]
}
```

Depois de configurar as bibliotecas do fornecedor, você precisará regenerar o carregador automático do seu aplicativo usando:

```
$ php composer.phar dump-autoload
```

Se você não estiver usando o Composer em seu aplicativo, precisará carregar manualmente todas as bibliotecas de fornecedores.

CAPÍTULO 31

Coleções

class Cake\Collection**Collection**

As classes de coleção fornecem um conjunto de ferramentas para manipular matrizes ou objetos `Traversable`. Se você já usou underscore.js, tem uma idéia do que pode esperar das classes de coleção.

Instâncias de coleção são imutáveis; modificar uma coleção irá gerar uma nova coleção. Isso torna o trabalho com objetos de coleção mais previsível, pois as operações são livres de efeitos colaterais.

Exemplo Rápido

Coleções podem ser criadas usando uma matriz ou um objeto `Traversable`. Você também interagirá com as coleções sempre que interagir com o ORM no CakePHP. Um simples uso de uma coleção seria:

```
use Cake\Collection\Collection;

$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);

// Crie uma nova coleção contendo elementos
// com um valor maior que um.
$overOne = $collection->filter(function ($value, $key, $iterator) {
    return $value > 1;
});
```

Você também pode usar a função auxiliar `collection()` em vez de `new Collection()`:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];

// Ambos formam uma instância de coleção.
```

(continues on next page)

(continuação da página anterior)

```
$collectionA = new Collection($items);
$collectionB = collection($items);
```

O benefício do método auxiliar é que é mais fácil encadear do que `(new Collection($items))`.

O `CollectionTrait` permite integrar recursos semelhantes a coleções em qualquer objeto `Traversable` que você possui no seu aplicativo.

Lista de Métodos

append	*appendItem*	*avg*
buffered	*chunk*	*chunkWithKeys*
combine	*compile*	*contains*
countBy	*each*	*every*
extract	*filter*	*first*
firstMatch	*groupBy*	*indexBy*
insert	*isEmpty*	*last*
listNested	*map*	*match*
max	*median*	*min*
nest	*prepend*	*prependItem*
reduce	*reject*	*sample*
shuffle	*skip*	*some*
sortBy	*stopWhen*	*sumOf*
take	*through*	*transpose*
unfold	*zip*	

Iterando

`Cake\Collection\Collection::`**`each`**`(`*$callback*`)`

As coleções podem ser iteradas e/ou transformadas em novas coleções com os métodos `each()` e `map()`. O método `each()` não criará uma nova coleção, mas permitirá que você modifique quaisquer objetos dentro da coleção:

```
$collection = new Collection($items);
$collection = $collection->each(function ($value, $key) {
    echo "Element $key: $value";
});
```

O retorno de `each()` será um objeto collection. Cada um iterará a coleção imediatamente aplicando o retorno de chamada a cada valor na coleção.

`Cake\Collection\Collection::`**`map`**`(`*$callback*`)`

O método `map()` criará uma nova coleção com base no retorno de chamada que está sendo aplicada a cada objeto na coleção original:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);
```

(continues on next page)

(continuação da página anterior)

```
$new = $collection->map(function ($value, $key) {
    return $value * 2;
});

// $result contém [2, 4, 6];
$result = $new->toList();

// $result contém ['a' => 2, 'b' => 4, 'c' => 6];
$result = $new->toArray();
```

O método `map()` criará um novo iterador que cria preguiçosamente os itens resultantes quando iterado.

`Cake\Collection\Collection::extract`(*$path*)

Um dos usos mais comuns de uma função `map()` é extrair uma única coluna de uma coleção. Se você deseja criar uma lista de elementos contendo os valores de uma propriedade específica, pode usar o método `extract()`:

```
$collection = new Collection($people);
$names = $collection->extract('name');

// $result contém ['mark', 'jose', 'barbara'];
$result = $names->toList();
```

Como em muitas outras funções da classe de coleção, você pode especificar um caminho separado por pontos para extrair colunas. Este exemplo retornará uma coleção que contém os nomes dos autores de uma lista de artigos:

```
$collection = new Collection($articles);
$names = $collection->extract('author.name');

// $result contém ['Maria', 'Stacy', 'Larry'];
$result = $names->toList();
```

Por fim, se a propriedade que você está procurando não pode ser expressa como um caminho, você pode usar uma função de retorno de chamada para retorná-la:

```
$collection = new Collection($articles);
$names = $collection->extract(function ($article) {
    return $article->author->name . ', ' . $article->author->last_name;
});
```

Freqüentemente, existem propriedades necessárias para extrair uma chave comum presente em várias matrizes ou objetos profundamente aninhados em outras estruturas. Para esses casos, você pode usar o combinador `{*}` na chave do caminho. Esse correspondente geralmente é útil ao combinar dados da associação HasMany e BelongsToMany:

```
$data = [
    [
        'name' => 'James',
        'phone_numbers' => [
            ['number' => 'number-1'],
            ['number' => 'number-2'],
            ['number' => 'number-3'],
        ]
    ],
    [
```

(continues on next page)

(continuação da página anterior)

```
        'name' => 'James',
        'phone_numbers' => [
            ['number' => 'number-4'],
            ['number' => 'number-5'],
        ]
    ]
];

$numbers = (new Collection($data))->extract('phone_numbers.{*}.number');
$numbers->toList();
// Retorna ['number-1', 'number-2', 'number-3', 'number-4', 'number-5']
```

Este último exemplo usa `toList()` diferente de outros exemplos, o que é importante quando estamos obtendo resultados com chaves possivelmente duplicadas. Ao usar `toList()`, garantimos a obtenção de todos os valores, mesmo que haja chaves duplicadas.

Ao contrário de *Cake\Utility\Hash::extract()* este método suporta apenas o curinga `{*}`. Todos os outros correspondentes de curinga e atributos não são suportados.

`Cake\Collection\Collection::`**`combine`**(*$keyPath*, *$valuePath*, *$groupPath = null*)

Coleções permitem que você crie uma nova coleção feita de chaves e valores em uma coleção existente. Os caminhos de chave e valor podem ser especificados com notação de caminhos com ponto:

```
$items = [
    ['id' => 1, 'name' => 'foo', 'parent' => 'a'],
    ['id' => 2, 'name' => 'bar', 'parent' => 'b'],
    ['id' => 3, 'name' => 'baz', 'parent' => 'a'],
];
$combined = (new Collection($items))->combine('id', 'name');

// O resultado ficará assim quando convertido em array
[
    1 => 'foo',
    2 => 'bar',
    3 => 'baz',
];
```

Opcionalmente, você também pode usar um `groupPath` para agrupar resultados com base em um caminho:

```
$combined = (new Collection($items))->combine('id', 'name', 'parent');

// O resultado ficará assim quando convertido em array
[
    'a' => [1 => 'foo', 3 => 'baz'],
    'b' => [2 => 'bar']
];
```

E por fim, você pode usar *closures* para criar caminhos de chaves/valores/grupos dinamicamente, por exemplo, ao trabalhar com entidades e datas (convertidas em instâncias `Cake/Time` pelo ORM), você pode querer agrupar os resultados por data:

```
$combined = (new Collection($entities))->combine(
    'id',
```

(continues on next page)

(continuação da página anterior)

```
    function ($entity) { return $entity; },
    function ($entity) { return $entity->date->toDateString(); }
);

// O resultado ficará assim quando convertido em array
[
    'date string like 2015-05-01' => ['entity1->id' => entity1, 'entity2->id' => entity2,
↪ ..., 'entityN->id' => entityN]
    'date string like 2015-06-01' => ['entity1->id' => entity1, 'entity2->id' => entity2,
↪ ..., 'entityN->id' => entityN]
]
```

Cake\Collection\Collection::**stopWhen**(*callable $c*)

Você pode parar a iteração a qualquer momento usando o método `stopWhen()`. A chamada em uma coleção criará uma nova e irá interromper a execução de novos resultados se a chamada passada retornar verdadeira para um dos elementos:

```
$items = [10, 20, 50, 1, 2];
$collection = new Collection($items);

$new = $collection->stopWhen(function ($value, $key) {
    // Pare no primeiro valor maior que 30
    return $value > 30;
});

// $result contém [10, 20];
$result = $new->toList();
```

Cake\Collection\Collection::**unfold**(*callable $callback*)

Às vezes, os itens internos de uma coleção contêm matrizes ou iteradores com mais itens. Se você deseja nivelar a estrutura interna para iterar uma vez todos os elementos, pode usar o método `unfold()`. Ele criará uma nova coleção que produzirá todos os elementos aninhados na coleção:

```
$items = [[1, 2, 3], [4, 5]];
$collection = new Collection($items);
$new = $collection->unfold();

// $result contém [1, 2, 3, 4, 5];
$result = $new->toList();
```

Ao passar uma chamada para `unfold()`, você pode controlar quais elementos serão desdobrados de cada item da coleção original. Isso é útil para retornar dados de serviços paginados:

```
$pages = [1, 2, 3, 4];
$collection = new Collection($pages);
$items = $collection->unfold(function ($page, $key) {
    // Um serviço da web imaginário que retorna uma página de resultados
    return MyService::fetchPage($page)->toList();
});

$allPagesItems = $items->toList();
```

Se você estiver usando o PHP 5.5+, você pode usar a palavra-chave `yield` dentro de `unfold()` para retornar quantos elementos de cada item da coleção você precisará:

```
$oddNumbers = [1, 3, 5, 7];
$collection = new Collection($oddNumbers);
$new = $collection->unfold(function ($oddNumber) {
    yield $oddNumber;
    yield $oddNumber + 1;
});

// $result contém [1, 2, 3, 4, 5, 6, 7, 8];
$result = $new->toList();
```

Cake\Collection\Collection::**chunk**(*$chunkSize*)

Ao lidar com grandes quantidades de itens em uma coleção, pode fazer sentido processar os elementos em lotes, em vez de um por um. Para dividir uma coleção em várias matrizes de um determinado tamanho, você pode usar a função chunk():

```
$items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
$collection = new Collection($items);
$chunked = $collection->chunk(2);
$chunked->toList(); // [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11]]
```

A função chunk é particularmente útil ao realizar o processamento em lote, por exemplo, com um resultado no banco de dados:

```
$collection = new Collection($articles);
$collection->map(function ($article) {
        // Alterar uma propriedade no artigo
        $article->property = 'changed';
    })
    ->chunk(20)
    ->each(function ($batch) {
        myBulkSave($batch); // Esta função será chamada para cada lote
    });
```

Cake\Collection\Collection::**chunkWithKeys**(*$chunkSize*)

Bem como *chunk()*, chunkWithKeys() permite dividir uma coleção em lotes menores, mas com as chaves preservadas. Isso é útil ao agrupar matrizes associativas:

```
$collection = new Collection([
    'a' => 1,
    'b' => 2,
    'c' => 3,
    'd' => [4, 5]
]);
$chunked = $collection->chunkWithKeys(2)->toList();
// Cria
[
    ['a' => 1, 'b' => 2],
    ['c' => 3, 'd' => [4, 5]]
]
```

Filtragem

Cake\Collection\Collection::**filter**(*$callback*)

As coleções facilitam a filtragem e a criação de novas coleções com base no resultado das funções de retorno de chamada. Você pode usar `filter()` para criar uma nova coleção de elementos que correspondem a um retorno de chamada de critério:

```
$collection = new Collection($people);
$ladies = $collection->filter(function ($person, $key) {
    return $person->gender === 'female';
});
$guys = $collection->filter(function ($person, $key) {
    return $person->gender === 'male';
});
```

Cake\Collection\Collection::**reject**(*callable $c*)

O inverso de `filter()` é `reject()`. Este método cria um filtro negativo, removendo elementos que correspondem à função de filtro:

```
$collection = new Collection($people);
$ladies = $collection->reject(function ($person, $key) {
    return $person->gender === 'male';
});
```

Cake\Collection\Collection::**every**(*$callback*)

Você pode fazer testes de verificação com funções de filtro. Para ver se todos os elementos de uma coleção correspondem a um teste, você pode usar `every()`:

```
$collection = new Collection($people);
$allYoungPeople = $collection->every(function ($person) {
    return $person->age < 21;
});
```

Cake\Collection\Collection::**some**(*$callback*)

Você pode ver se a coleção contém pelo menos um elemento que corresponde a uma função de filtro usando o método `some()`:

```
$collection = new Collection($people);
$hasYoungPeople = $collection->some(function ($person) {
    return $person->age < 21;
});
```

Cake\Collection\Collection::**match**(*$conditions*)

Se você precisar extrair uma nova coleção contendo apenas os elementos que contêm um determinado conjunto de propriedades, use o método `match()`:

```
$collection = new Collection($comments);
$commentsFromMark = $collection->match(['user.name' => 'Mark']);
```

Cake\Collection\Collection::**firstMatch**(*$conditions*)

O nome da propriedade pode ser um caminho separado por pontos. Você pode percorrer entidades aninhadas e verificar valores que elas contêm. Quando você só precisa do primeiro elemento correspondente de uma coleção, pode usar `firstMatch()`:

```
$collection = new Collection($comments);
$comment = $collection->firstMatch([
    'user.name' => 'Mark',
    'active' => true
]);
```

Como você pode ver acima, `match()` e `firstMatch()` permitem fornecer várias condições para a correspondência. Além disso, as condições podem ser para caminhos diferentes, permitindo expressar condições complexas para comparação.

Agregação

Cake\Collection\Collection::**reduce**(*$callback*)

A contraparte de uma operação `map()` geralmente é uma `reduce`. Esta função ajudará você a criar um único resultado de todos os elementos em uma coleção:

```
$totalPrice = $collection->reduce(function ($accumulated, $orderLine) {
    return $accumulated + $orderLine->price;
}, 0);
```

No exemplo acima, `$totalPrice` será a soma de todos os preços únicos contidos na coleção. Observe que o segundo argumento para da função `reduce()` assume o valor inicial da operação de redução que você está executando:

```
$allTags = $collection->reduce(function ($accumulated, $article) {
    return array_merge($accumulated, $article->tags);
}, []);
```

Cake\Collection\Collection::**min**(*string|$callback*, *$type = SORT_NUMERIC*)

Para extrair o valor mínimo de uma coleção com base em uma propriedade, basta usar a função `min()`. Isso retornará o elemento completo da coleção e não apenas o menor valor encontrado:

```
$collection = new Collection($people);
$youngest = $collection->min('age');

echo $youngest->name;
```

Você também pode expressar a propriedade para comparar, fornecendo um caminho ou uma função de retorno de chamada:

```
$collection = new Collection($people);
$personYoungestChild = $collection->min(function ($person) {
    return $person->child->age;
});

$personWithYoungestDad = $collection->min('dad.age');
```

Cake\Collection\Collection::**max**(*$callback*, *$type = SORT_NUMERIC*)

O mesmo pode ser aplicado à função `max()`, que retornará um único elemento da coleção com o maior valor de propriedade:

```
$collection = new Collection($people);
$oldest = $collection->max('age');

$personOldestChild = $collection->max(function ($person) {
    return $person->child->age;
});

$personWithOldestDad = $collection->max('dad.age');
```

Cake\Collection\Collection::**sumOf**(*$path = null*)

Finalmente, o método `sumOf()` retornará a soma de uma propriedade de todos os elementos:

```
$collection = new Collection($people);
$sumOfAges =  $collection->sumOf('age');

$sumOfChildrenAges = $collection->sumOf(function ($person) {
    return $person->child->age;
});

$sumOfDadAges = $collection->sumOf('dad.age');
```

Cake\Collection\Collection::**avg**(*$path = null*)

Calcule o valor médio dos elementos na coleção. Opcionalmente, forneça um caminho correspondente ou função para extrair valores e gerar a média:

```
$items = [
   ['invoice' => ['total' => 100]],
   ['invoice' => ['total' => 200]],
];

// Média: 150
$average = (new Collection($items))->avg('invoice.total');
```

Cake\Collection\Collection::**median**(*$path = null*)

Calcule o valor mediano de um conjunto de elementos. Opcionalmente, poderá fornecer um caminho correspondente ou função para extrair valores para gerar a mediana:

```
$items = [
  ['invoice' => ['total' => 400]],
  ['invoice' => ['total' => 500]],
  ['invoice' => ['total' => 100]],
  ['invoice' => ['total' => 333]],
  ['invoice' => ['total' => 200]],
];

// Média: 333
$median = (new Collection($items))->median('invoice.total');
```

Agrupamento e Contagem

Cake\Collection\Collection::**groupBy**(*$callback*)

Os valores da coleção podem ser agrupados por chaves diferentes em uma nova coleção quando eles compartilham o mesmo valor para uma propriedade:

```
$students = [
    ['name' => 'Mark', 'grade' => 9],
    ['name' => 'Andrew', 'grade' => 10],
    ['name' => 'Stacy', 'grade' => 10],
    ['name' => 'Barbara', 'grade' => 9]
];
$collection = new Collection($students);
$studentsByGrade = $collection->groupBy('grade');

// O resultado ficará assim quando convertido em array:
[
  10 => [
    ['name' => 'Andrew', 'grade' => 10],
    ['name' => 'Stacy', 'grade' => 10]
  ],
  9 => [
    ['name' => 'Mark', 'grade' => 9],
    ['name' => 'Barbara', 'grade' => 9]
  ]
]
```

Como de costume, é possível fornecer um caminho separado por pontos para propriedades aninhadas ou sua própria função de retorno de chamada para gerar os grupos dinamicamente:

```
$commentsByUserId = $comments->groupBy('user.id');

$classResults = $students->groupBy(function ($student) {
    return $student->grade > 6 ? 'approved' : 'denied';
});
```

Cake\Collection\Collection::**countBy**(*$callback*)

Se você deseja apenas saber o número de ocorrências por grupo, pode fazê-lo usando o método `countBy()`. Ele usa os mesmos argumentos de `groupBy`, portanto já deve ser familiar para você:

```
$classResults = $students->countBy(function ($student) {
    return $student->grade > 6 ? 'approved' : 'denied';
});

// O resultado ficará assim quando convertido em array:
['approved' => 70, 'denied' => 20]
```

Cake\Collection\Collection::**indexBy**(*$callback*)

Em certos casos, você sabe que um elemento é exclusivo para a propriedade que você deseja agrupar. Se você deseja um único resultado por grupo, pode usar a função `indexBy()`:

```
$usersById = $users->indexBy('id');

// Quando convertido em resultado da matriz, pode parecer
[
    1 => 'markstory',
    3 => 'jose_zap',
    4 => 'jrbasso'
]
```

Assim como na função `groupBy()`, você também pode usar um caminho de propriedade ou um retorno de chamada:

```
$articlesByAuthorId = $articles->indexBy('author.id');

$filesByHash = $files->indexBy(function ($file) {
    return md5($file);
});
```

Cake\Collection\Collection::**zip**(*$items*)

Os elementos de diferentes coleções podem ser agrupados usando o método `zip()`. Ele retornará uma nova coleção contendo uma matriz que agrupa os elementos de cada coleção que são colocados na mesma posição:

```
$odds = new Collection([1, 3, 5]);
$pairs = new Collection([2, 4, 6]);
$combined = $odds->zip($pairs)->toList(); // [[1, 2], [3, 4], [5, 6]]
```

Você também pode compactar várias coleções de uma só vez:

```
$years = new Collection([2013, 2014, 2015, 2016]);
$salaries = [1000, 1500, 2000, 2300];
$increments = [0, 500, 500, 300];

$rows = $years->zip($salaries, $increments)->toList();
// Retorna:
[
    [2013, 1000, 0],
    [2014, 1500, 500],
    [2015, 2000, 500],
    [2016, 2300, 300]
]
```

Como você já pode ver, o método `zip()` é muito útil para transpor matrizes multidimensionais:

```
$data = [
    2014 => ['jan' => 100, 'feb' => 200],
    2015 => ['jan' => 300, 'feb' => 500],
    2016 => ['jan' => 400, 'feb' => 600],
]

// Reunindo dados de janeiro e fevereiro

$firstYear = new Collection(array_shift($data));
$firstYear->zip($data[0], $data[1])->toList();
```

(continues on next page)

(continuação da página anterior)

```
// Ou $firstYear->zip(...$data) em PHP >= 5.6

// Retorna
[
    [100, 300, 400],
    [200, 500, 600]
]
```

Ordenação

`Cake\Collection\Collection::sortBy`(*$callback*, *$order = SORT_DESC*, *$sort = SORT_NUMERIC*)

Os valores da coleção podem ser classificados em ordem crescente ou decrescente com base em uma coluna ou função personalizada. Para criar uma nova coleção classificada com os valores de outra, você pode usar `sortBy`:

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age');
```

Como visto acima, você pode classificar passando o nome de uma coluna ou propriedade presente nos valores da coleção. Você também pode especificar um caminho de propriedade usando a notação de ponto. O próximo exemplo classificará os artigos pelo nome do autor:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('author.name');
```

O método `sortBy()` é flexível o suficiente para permitir que você especifique uma função extratora que permitirá selecionar dinamicamente o valor a ser usado para comparar dois valores diferentes na coleção:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy(function ($article) {
    return $article->author->name . '-' . $article->title;
});
```

Para especificar em qual direção a coleção deve ser classificada, é necessário fornecer `SORT_ASC` ou `SORT_DESC` como o segundo parâmetro para classificar na direção ascendente ou descendente, respectivamente. Por padrão, as coleções são classificadas em direção descendente:

```
$collection = new Collection($people);
$sorted = $collection->sortBy('age', SORT_ASC);
```

Às vezes, você precisará especificar que tipo de dados você está tentando comparar para obter resultados consistentes. Para esse fim, você deve fornecer um terceiro argumento na função `sortBy()` com uma das seguintes constantes:

- **SORT_NUMERIC**: Para comparar números
- **SORT_STRING**: Para comparar valores de sequência
- **SORT_NATURAL**: Para classificar sequência contendo números e se você desejar que esses números sejam ordenados de maneira natural. Por exemplo: mostrando «10» depois de «2».
- **SORT_LOCALE_STRING**: Para comparar seqüências de caracteres com base na localidade atual.

Por padrão, `SORT_NUMERIC` é usado:

```
$collection = new Collection($articles);
$sorted = $collection->sortBy('title', SORT_ASC, SORT_NATURAL);
```

Aviso: Muitas vezes, é caro iterar coleções classificadas mais de uma vez. Se você planeja fazer isso, considere converter a coleção em uma matriz ou simplesmente use o método `compile()` nela.

Trabalhando com dados em Árvore

`Cake\Collection\Collection::nest`(*$idPath*, *$parentPath*)

Nem todos os dados devem ser representados de maneira linear. As coleções facilitam a construção e o nivelamento de estruturas hierárquicas ou aninhadas. Criar uma estrutura aninhada em que os filhos são agrupados por uma propriedade de identificador pai é fácil com o método `nest()`.

Dois parâmetros são necessários para esta função. O primeiro é a propriedade que representa o identificador do item. O segundo parâmetro é o nome da propriedade que representa o identificador para o item pai:

```
$collection = new Collection([
    ['id' => 1, 'parent_id' => null, 'name' => 'Birds'],
    ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds'],
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish'],
    ['id' => 6, 'parent_id' => null, 'name' => 'Fish'],
]);

$collection->nest('id', 'parent_id')->toList();
// Retorna
[
    [
        'id' => 1,
        'parent_id' => null,
        'name' => 'Birds',
        'children' => [
            ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds', 'children' => []],
            ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle', 'children' => []],
            ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull', 'children' => []],
        ]
    ],
    [
        'id' => 6,
        'parent_id' => null,
        'name' => 'Fish',
        'children' => [
            ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish', 'children' => []],
        ]
    ]
];
```

Os elementos filhos são aninhados dentro da propriedade `children` dentro de cada um dos itens da coleção. Esse tipo de representação de dados é útil para renderizar menus ou percorrer elementos até um determinado nível na árvore.

Cake\Collection\Collection::**listNested**(*$order = 'desc'*, *$nestingKey = 'children'*)

O inverso de `nest()` é `listNested()`. Este método permite nivelar uma estrutura de árvore novamente em uma estrutura linear. São necessários dois parâmetros; o primeiro é o modo de deslocamento (asc, desc ou leaves) e o segundo é o nome da propriedade que contém os filhos de cada elemento da coleção.

Tomando a entrada da coleção aninhada criada no exemplo anterior, podemos deixar nivelado:

```
$nested->listNested()->toList();

// Retorna
[
    ['id' => 1, 'parent_id' => null, 'name' => 'Birds', 'children' => [...]],
    ['id' => 2, 'parent_id' => 1, 'name' => 'Land Birds'],
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 6, 'parent_id' => null, 'name' => 'Fish', 'children' => [...]],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish']
]
```

Por padrão, a árvore é atravessada da raiz para as extremidades. Você também pode instruí-lo a retornar apenas os elementos filhos da árvore:

```
$nested->listNested()->toList();

// Retorna
[
    ['id' => 3, 'parent_id' => 1, 'name' => 'Eagle'],
    ['id' => 4, 'parent_id' => 1, 'name' => 'Seagull'],
    ['id' => 5, 'parent_id' => 6, 'name' => 'Clown Fish']
]
```

Depois de converter uma árvore em uma lista aninhada, você pode usar o método `printer()` para configurar como a saída da lista deve ser formatada:

```
$nested->listNested()->printer('name', 'id', '--')->toArray();

// Retorna
[
    3 => 'Eagle',
    4 => 'Seagull',
    5 -> '--Clown Fish',
]
```

O método `printer()` também permite usar um retorno de chamada para gerar as chaves e/ou valores:

```
$nested->listNested()->printer(
    function ($el) {
        return $el->name;
    },
    function ($el) {
        return $el->id;
    }
);
```

Outros Métodos

`Cake\Collection\Collection::isEmpty()`

Permite que você veja se uma coleção contém algum elemento:

```
$collection = new Collection([]);
// Retorna true
$collection->isEmpty();

$collection = new Collection([1]);
// Retorna false
$collection->isEmpty();
```

`Cake\Collection\Collection::contains($value)`

As coleções permitem que você verifique rapidamente se elas contêm um valor específico usando o método `contains()`:

```
$items = ['a' => 1, 'b' => 2, 'c' => 3];
$collection = new Collection($items);
$hasThree = $collection->contains(3);
```

As comparações são realizadas usando o operador ===. Se você deseja fazer tipos de comparação mais flexíveis, pode usar o método `some()`.

`Cake\Collection\Collection::shuffle()`

Às vezes, você pode querer mostrar uma coleção de valores em uma ordem aleatória. Para criar uma nova coleção que retornará cada valor em uma posição diferente, use o `shuffle`:

```
$collection = new Collection(['a' => 1, 'b' => 2, 'c' => 3]);

// Isso poderia retornar [2, 3, 1]
$collection->shuffle()->toList();
```

`Cake\Collection\Collection::transpose()`

Ao transpor uma coleção, você obtém uma nova coleção contendo uma linha feita de cada uma das colunas originais:

```
$items = [
   ['Products', '2012', '2013', '2014'],
   ['Product A', '200', '100', '50'],
   ['Product B', '300', '200', '100'],
   ['Product C', '400', '300', '200'],
]
$transpose = (new Collection($items))->transpose()->toList();

// Retorna
[
    ['Products', 'Product A', 'Product B', 'Product C'],
    ['2012', '200', '300', '400'],
    ['2013', '100', '200', '300'],
    ['2014', '50', '100', '200'],
]
```

Retirando Elementos

Cake\Collection\Collection::**sample**(*$length = 10*)

Baralhar uma coleção geralmente é útil ao fazer análises estatísticas rápidas. Outra operação comum ao executar esse tipo de tarefa é retirar alguns valores aleatórios de uma coleção, para que mais testes possam ser realizados. Por exemplo, se você quiser selecionar 5 usuários aleatórios aos quais deseja aplicar alguns testes A/B, poderá usar a função `sample()`:

```
$collection = new Collection($people);

// Retire no máximo 20 usuários aleatórios desta coleção
$testSubjects = $collection->sample(20);
```

`sample()` terá no máximo o número de valores que você especificar no primeiro argumento. Se não houver elementos suficientes na coleção para satisfazer a amostra, a coleção completa em uma ordem aleatória será retornada.

Cake\Collection\Collection::**take**(*$length*, *$offset*)

Sempre que você desejar obter uma fatia de uma coleção, use a função `take()`, ela criará uma nova coleção com, no máximo, o número de valores que você especificar no primeiro argumento, iniciando na posição passada no segundo argumento:

```
$topFive = $collection->sortBy('age')->take(5);

// Leve 5 pessoas da coleção a partir da posição 4
$nextTopFive = $collection->sortBy('age')->take(5, 4);
```

As posições são baseadas em zero, portanto, o número da primeira posição é `0`.

Cake\Collection\Collection::**skip**(*$length*)

Embora o segundo argumento de `take()` possa ajudá-lo a pular alguns elementos antes de obtê-los da coleção, você também pode usar `skip()` para o mesmo objetivo que uma maneira de tirar o resto dos elementos depois de uma certa posição:

```
$collection = new Collection([1, 2, 3, 4]);
$allExceptFirstTwo = $collection->skip(2)->toList(); // [3, 4]
```

Cake\Collection\Collection::**first**()

Um dos usos mais comuns de `take()` é obter o primeiro elemento da coleção. Um método de atalho para alcançar o mesmo objetivo é usar o método `first()`:

```
$collection = new Collection([5, 4, 3, 2]);
$collection->first(); // Returns 5
```

Cake\Collection\Collection::**last**()

Da mesma forma, você pode obter o último elemento de uma coleção usando o método `last()`:

```
$collection = new Collection([5, 4, 3, 2]);
$collection->last(); // Retorna 2
```

Expansão de Coleções

Cake\Collection\Collection::**append**(*array|Traversable $items*)

Você pode compor várias coleções em uma única. Isso permite coletar dados de várias fontes, concatená-los e aplicar outras funções de coleta de maneira muito suave. O método `append()` retornará uma nova coleção contendo os valores das duas fontes:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->append($phpTweets);

// Tweets contendo cakefest de ambas as fontes
$myTimeline->filter(function ($tweet) {
    return strpos($tweet, 'cakefest');
});
```

Cake\Collection\Collection::**appendItem**(*$value*, *$key*)

Permite anexar um item com uma chave opcional à coleção. Se você especificar uma chave que já existe na coleção, o valor não será substituído:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->appendItem($newTweet, 99);
```

Cake\Collection\Collection::**prepend**(*$items*)

O método `prepend()` retornará uma nova coleção contendo os valores das duas fontes:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->prepend($phpTweets);
```

Cake\Collection\Collection::**prependItem**(*$value*, *$key*)

Permite anexar um item com uma chave opcional à coleção. Se você especificar uma chave que já existe na coleção, o valor não será substituído:

```
$cakephpTweets = new Collection($tweets);
$myTimeline = $cakephpTweets->prependItem($newTweet, 99);
```

Aviso: Ao anexar de fontes diferentes, você pode esperar que algumas chaves de ambas as coleções sejam iguais. Por exemplo, ao anexar duas matrizes simples. Isso pode apresentar um problema ao converter uma coleção em uma matriz usando `toArray()`. Se você não deseja que os valores de uma coleção substituam os outros na coleção anterior com base em sua chave, certifique-se de chamar `toList()` para soltar as chaves e preservar todos os valores.

Elementos de Modificação

Cake\Collection\Collection::**insert**(*$path*, *$items*)

Às vezes, você pode ter dois conjuntos de dados separados que gostaria de inserir os elementos de um conjunto em cada um dos elementos do outro conjunto. Este é um caso muito comum quando você busca dados de uma fonte que não oferece suporte à mesclagem de dados ou se une nativamente.

As coleções oferecem um método `insert()` que permitirá inserir cada um dos elementos em uma coleção em uma propriedade dentro de cada um dos elementos de outra coleção:

```
$users = [
    ['username' => 'mark'],
    ['username' => 'juan'],
    ['username' => 'jose']
];

$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript'],
    ['Javascript', 'Prolog']
];

$merged = (new Collection($users))->insert('skills', $languages);
```

Quando convertida em uma matriz, a coleção `$merged` ficará assim:

```
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => ['Javascript', 'Prolog']]
];
```

O primeiro parâmetro para o método `insert()` é um caminho de propriedades separado por pontos a seguir para que os elementos possam ser inseridos nessa posição. O segundo argumento é qualquer coisa que possa ser convertida em um objeto de coleção.

Observe que os elementos são inseridos pela posição em que foram encontrados, portanto, o primeiro elemento da segunda coleção é mesclado no primeiro elemento da primeira coleção.

Se não houver elementos suficientes na segunda coleção para inserir na primeira, a propriedade target será preenchida com valores `null`:

```
$languages = [
    ['PHP', 'Python', 'Ruby'],
    ['Bash', 'PHP', 'Javascript']
];

$merged = (new Collection($users))->insert('skills', $languages);

// Irá fornecer
[
    ['username' => 'mark', 'skills' => ['PHP', 'Python', 'Ruby']],
    ['username' => 'juan', 'skills' => ['Bash', 'PHP', 'Javascript']],
    ['username' => 'jose', 'skills' => null]
];
```

O método `insert()` pode operar elementos ou objetos da matriz implementando a interface `ArrayAccess`.

Tornando Reutilizáveis os Métodos de Coleta

Usar closures para métodos de coleta é ótimo quando o trabalho a ser feito é pequeno e focado, mas pode ficar confuso muito rapidamente. Isso se torna mais óbvio quando muitos métodos diferentes precisam ser chamados ou quando o comprimento dos métodos de closures é superior a apenas algumas linhas.

Também existem casos em que a lógica usada para os métodos de coleta pode ser reutilizada em várias partes do seu aplicativo. É recomendável considerar a extração de lógica de coleção complexa para separar classes. Por exemplo, imagine uma closure longa como esta:

```
$collection
        ->map(function ($row, $key) {
            if (!empty($row['items'])) {
                $row['total'] = collection($row['items'])->sumOf('price');
            }

            if (!empty($row['total'])) {
                $row['tax_amount'] = $row['total'] * 0.25;
            }

            // Mais código aqui...

            return $modifiedRow;
        });
```

Isso pode ser refatorado criando outra classe:

```
class TotalOrderCalculator
{
        public function __invoke($row, $key)
        {
            if (!empty($row['items'])) {
                $row['total'] = collection($row['items'])->sumOf('price');
            }

            if (!empty($row['total'])) {
                $row['tax_amount'] = $row['total'] * 0.25;
            }

            // Mais código aqui...

            return $modifiedRow;
        }
}

// Usa a lógica em sua chamada de map()
$collection->map(new TotalOrderCalculator)
```

`Cake\Collection\Collection::through(`*$callback*`)`

Às vezes, uma cadeia de chamadas de método de coleção pode se tornar reutilizável em outras partes do seu aplicativo, mas apenas se elas forem chamadas nessa ordem específica. Nesses casos, você pode usar `through()` em combinação

com uma classe implementando `__invoke` para distribuir suas chamadas úteis de processamento de dados:

```
$collection
        ->map(new ShippingCostCalculator)
        ->map(new TotalOrderCalculator)
        ->map(new GiftCardPriceReducer)
        ->buffered()
       ...
```

As chamadas de método acima podem ser extraídas para uma nova classe, para que não precisem ser repetidas sempre:

```
class FinalCheckOutRowProcessor
{
        public function __invoke($collection)
        {
                return $collection
                        ->map(new ShippingCostCalculator)
                        ->map(new TotalOrderCalculator)
                        ->map(new GiftCardPriceReducer)
                        ->buffered()
                       ...
        }
}

// Agora você pode usar o método through() para chamar todos os métodos de uma só vez
$collection->through(new FinalCheckOutRowProcessor);
```

Otimizando Coleções

`Cake\Collection\Collection::buffered()`

As coleções geralmente executam a maioria das operações que você cria usando suas funções de maneira lenta. Isso significa que, embora você possa chamar uma função, isso não significa que ela seja executada imediatamente. Isso é verdade para muitas funções nesta classe. A avaliação lenta permite economizar recursos em situações em que você não usa todos os valores em uma coleção. Você não pode usar todos os valores quando a iteração parar mais cedo ou quando um caso de exceção/falha for atingido mais cedo.

Além disso, a avaliação lenta ajuda a acelerar algumas operações. Considere o seguinte exemplo:

```
$collection = new Collection($oneMillionItems);
$collection = $collection->map(function ($item) {
    return $item * 2;
});
$itemsToShow = $collection->take(30);
```

Se as coleções não tivessem sido preguiçosas, teríamos executado um milhão de operações, embora desejássemos mostrar apenas 30 elementos. Em vez disso, nossa operação de mapa foi aplicada apenas aos 30 elementos que usamos. Também podemos obter benefícios dessa avaliação preguiçosa para coleções menores quando fazemos mais de uma operação nelas. Por exemplo: chamando `map()` duas vezes e depois `filter()`.

A avaliação preguiçosa também traz sua desvantagem. Você pode estar executando as mesmas operações mais de uma vez se otimizar uma coleção prematuramente. Considere este exemplo:

```
$ages = $collection->extract('age');

$youngerThan30 = $ages->filter(function ($item) {
    return $item < 30;
});

$olderThan30 = $ages->filter(function ($item) {
    return $item > 30;
});
```

Se iterarmos `youngerThan30` e `olderThan30`, infelizmente a coleção executaria a operação `extract()` duas vezes. Isso ocorre porque as coleções são imutáveis e a operação de extração lenta é feita para os dois filtros.

Felizmente, podemos superar esse problema com uma única função. Se você planeja reutilizar os valores de determinadas operações mais de uma vez, é possível compilar os resultados em outra coleção usando a função `buffered()`:

```
$ages = $collection->extract('age')->buffered();
$youngerThan30 = ...
$olderThan30 = ...
```

Agora, quando as duas coleções forem iteradas, elas chamarão a operação de extração apenas uma vez.

Tornando as Coleções Rebobináveis

O método `buffered()` também é útil para converter iteradores não-rebobináveis em coleções que podem ser iteradas mais de uma vez:

```
// Em PHP 5.5+
public function results()
{
    ...
    foreach ($transientElements as $e) {
        yield $e;
    }
}
$rewindable = (new Collection(results()))->buffered();
```

Coleções de Clonagem

`Cake\Collection\Collection::compile(`*$preserveKeys = true*`)`

Às vezes, você precisa obter um clone dos elementos de outra coleção. Isso é útil quando você precisa repetir o mesmo conjunto de locais diferentes ao mesmo tempo. Para clonar uma coleção de outra, use o método `compile()`:

```
$ages = $collection->extract('age')->compile();

foreach ($ages as $age) {
    foreach ($collection as $element) {
        echo h($element->name) . ' - ' . $age;
    }
}
```

CAPÍTULO 32

Pasta & Arquivo

Os utilitários de pasta e arquivo são classes convenientes para ajudá-lo a ler e gravar/anexar arquivos, listar arquivos dentro de uma pasta e outras tarefas comuns relacionadas ao diretório.

Obsoleto desde a versão 4.0: As classes `File` e `Folder` serão removidas na 5.0. Use classes SPL como `SplFileInfo` ou `SplFileObject` e classes iterator como `RecursiveDirectoryIterator`, `RecursiveRegexIterator` etc.

Uso Básico

Certifique-se de que as classes estejam carregadas:

```
use Cake\Filesystem\Folder;
use Cake\Filesystem\File;
```

Com isso podemos configurar uma nova instância da pasta:

```
$dir = new Folder('/path/to/folder');
```

e então pesquise todos os arquivos *.php* dentro dessa pasta usando regex:

```
$files = $dir->find('.*\.php');
```

Agora podemos percorrer os arquivos e ler ou escrever/anexar ao conteúdo ou simplesmente excluir o arquivo:

```
foreach ($files as $file) {
    $file = new File($dir->pwd() . DS . $file);
    $contents = $file->read();
    // $file->write('Estou substituindo o conteúdo deste arquivo');
    // $file->append('Estou adicionando ao final deste arquivo.');
    // $file->delete(); // Estou excluindo este arquivo
```

(continues on next page)

(continuação da página anterior)

```
    $file->close(); // Certifique-se de fechar o arquivo quando terminar
}
```

API Pastas

class Cake\Filesystem**Folder**(*string $path = false*, *boolean $create = false*, *string|boolean $mode = false*)

```
// Cria uma nova pasta com as permissões 0755
$dir = new Folder('/path/to/folder', true, 0755);
```

property Cake\Filesystem\Folder::**$path**

Caminho da pasta atual. *Folder::pwd()* retornará a mesma informação.

property Cake\Filesystem\Folder::**$sort**

Se os resultados da lista devem ou não ser classificados por nome.

property Cake\Filesystem\Folder::**$mode**

Modo a ser usado ao criar pastas. O padrão é `0755`. Não faz nada em máquinas Windows.

static Cake\Filesystem\Folder::**addPathElement**(*string $path*, *string $element*)

Retorna $path com $elemento adicionado, com a barra correta:

```
$path = Folder::addPathElement('/a/path/for', 'testing');
// $path é igual a /a/path/for/testing
```

$element também pode ser um array:

```
$path = Folder::addPathElement('/a/path/for', ['testing', 'another']);
// $path é igual a /a/path/for/testing/another
```

Cake\Filesystem\Folder::**cd**(*$path*)

Mude o diretório para $path. Retorna `false` em caso de falha:

```
$folder = new Folder('/foo');
echo $folder->path; // Exibe /foo
$folder->cd('/bar');
echo $folder->path; // Exibe /bar
$false = $folder->cd('/non-existent-folder');
```

Cake\Filesystem\Folder::**chmod**(*string $path*, *integer $mode = false*, *boolean $recursive = true*, *array $exceptions = []*)

Altere o modo em uma estrutura de diretório recursivamente. Isso inclui alterar o modo dos arquivos também:

```
$dir = new Folder();
$dir->chmod('/path/to/folder', 0755, true, ['skip_me.php']);
```

Cake\Filesystem\Folder::**copy**(*array|string $options = []*)

Copie recursivamente um diretório. O único parâmetro $options pode ser um caminho para a cópia ou um conjunto de opções:

```
$folder1 = new Folder('/path/to/folder1');
$folder1->copy('/path/to/folder2');
// Colocará a pasta1 e todo o seu conteúdo na pasta2

$folder = new Folder('/path/to/folder');
$folder->copy([
    'to' => '/path/to/new/folder',
    'from' => '/path/to/copy/from', // Irá causar a ocorrência de um cd()
    'mode' => 0755,
    'skip' => ['skip-me.php', '.git'],
    'scheme' => Folder::SKIP  // Pule diretórios/arquivos que já existem.
]);
```

Existem 3 esquemas suportados:

- `Folder::SKIP` pule a cópia/movimentação de arquivos e diretórios que existem no diretório de destino.
- `Folder::MERGE` mescla os diretórios de origem/destino. Os arquivos no diretório de origem substituirão os arquivos no diretório de destino. O conteúdo do diretório será mesclado.
- `Folder::OVERWRITE` sobrescreve os arquivos e diretórios existentes no diretório de destino pelos do diretório de origem. Se ambos contiverem o mesmo subdiretório, o conteúdo do diretório de destino será removido e substituído pelo de origem.

static Cake\Filesystem\Folder::**correctSlashFor**(*string $path*)

Retorna um conjunto correto de barras para o $path fornecido ("\" para caminhos do Windows e "/" para outros caminhos).

Cake\Filesystem\Folder::**create**(*string $pathname*, *integer $mode = false*)

Crie uma estrutura de diretório recursivamente. Pode ser usado para criar estruturas de caminho mais profundo como */foo/bar/baz/shoe/horn*:

```
$folder = new Folder();
if ($folder->create('foo' . DS . 'bar' . DS . 'baz' . DS . 'shoe' . DS . 'horn')) {
    // As pastas aninhadas foram criadas com sucesso
}
```

Cake\Filesystem\Folder::**delete**(*string $path = null*)

Remova diretórios recursivamente se o sistema permitir:

```
$folder = new Folder('foo');
if ($folder->delete()) {
    // Foo foi excluído com sucesso e também suas pastas aninhadas
}
```

Cake\Filesystem\Folder::**dirsize**()

Retorna o tamanho em bytes desta pasta e seu conteúdo.

Cake\Filesystem\Folder::**errors**()

Obtenha o erro do método mais recente.

Cake\Filesystem\Folder::**find**(*string $regexpPattern = '.*'*, *boolean $sort = false*)

Retorna uma matriz de todos os arquivos correspondentes no diretório atual:

```
// Encontre todos os .png em sua pasta webroot/img/ e classifique os resultados
$dir = new Folder(WWW_ROOT . 'img');
$files = $dir->find('.*\.png', true);
/*
Array
(
    [0] => cake.icon.png
    [1] => test-error-icon.png
    [2] => test-fail-icon.png
    [3] => test-pass-icon.png
    [4] => test-skip-icon.png
)
*/
```

Nota: Os métodos find e findRecursive da pasta só encontrarão arquivos. Se você gostaria de obter pastas e arquivos, consulte *Folder::read()* ou *Folder::tree()*

Cake\Filesystem\Folder::**findRecursive**(*string $pattern = '.*', boolean $sort = false*)

Retorna uma matriz de todos os arquivos correspondentes dentro e abaixo do diretório atual:

```
// Encontre arquivos recursivamente começando com teste ou índice
$dir = new Folder(WWW_ROOT);
$files = $dir->findRecursive('(test|index).*');
/*
Array
(
    [0] => /var/www/cake/webroot/index.php
    [1] => /var/www/cake/webroot/test.php
    [2] => /var/www/cake/webroot/img/test-skip-icon.png
    [3] => /var/www/cake/webroot/img/test-fail-icon.png
    [4] => /var/www/cake/webroot/img/test-error-icon.png
    [5] => /var/www/cake/webroot/img/test-pass-icon.png
)
*/
```

Cake\Filesystem\Folder::**inCakePath**(*string $path = ''*)

Retorna `true` se o arquivo está em um determinado CakePath.

Cake\Filesystem\Folder::**inPath**(*string $path = '', boolean $reverse = false*)

Retorna `true` se o arquivo está no caminho fornecido:

```
$Folder = new Folder(WWW_ROOT);
$result = $Folder->inPath(APP);
// $result = false, /var/www/example/src/ não está em /var/www/example/webroot/

$result = $Folder->inPath(WWW_ROOT . 'img' . DS, true);
// $result = true, /var/www/example/webroot/img/ está em /var/www/example/webroot/
```

static Cake\Filesystem\Folder::**isAbsolute**(*string $path*)

Retorna `true` se o $path fornecido for um caminho absoluto.

static Cake\Filesystem\Folder::**isSlashTerm**(*string $path*)

Retorna `true` se o $path termina em uma barra (ou seja, termina com uma barra):

```
$result = Folder::isSlashTerm('/my/test/path');
// $result = false
$result = Folder::isSlashTerm('/my/test/path/');
// $result = true
```

static Cake\Filesystem\Folder::**isWindowsPath**(*string $path*)

Retorna `true` se o $path fornecido for um caminho do Windows.

Cake\Filesystem\Folder::**messages**()

Obtenha as mensagens do método mais recente.

Cake\Filesystem\Folder::**move**(*array $options*)

Move recursivamente o diretório.

static Cake\Filesystem\Folder::**normalizeFullPath**(*string $path*)

Retorna um caminho com barras normalizadas para o sistema operacional.

Cake\Filesystem\Folder::**pwd**()

Retorna o caminho atual

Cake\Filesystem\Folder::**read**(*boolean $sort = true*, *array|boolean $exceptions = false*, *boolean $fullPath = false*)

Retorna uma matriz do conteúdo do diretório atual. A matriz retornada contém duas submatrizes e uma de diretórios e uma de arquivos:

```
$dir = new Folder(WWW_ROOT);
$files = $dir->read(true, ['files', 'index.php']);
/*
Array
(
    [0] => Array // Folders
        (
            [0] => css
            [1] => img
            [2] => js
        )
    [1] => Array // Files
        (
            [0] => .htaccess
            [1] => favicon.ico
            [2] => test.php
        )
)
*/
```

Cake\Filesystem\Folder::**realpath**(*string $path*)

Pegue o caminho real (levando «..» em consideração).

static Cake\Filesystem\Folder::**slashTerm**(*string $path*)

Retorna $path com barra de terminação adicionada (corrigido para Windows ou outro sistema operacional).

Cake\Filesystem\Folder::**tree**(*null|string $path = null*, *array|boolean $exceptions = true*, *null|string $type = null*)

Retorna uma matriz de diretórios e arquivos aninhados em cada diretório.

API de Arquivos

class Cake\Filesystem**File**(*string $path*, *boolean $create = false*, *integer $mode = 755*)

```
// Cria um novo arquivo com as permissões 0644
$file = new File('/path/to/file.php', true, 0644);
```

property Cake\Filesystem\File::$**Folder**

O objeto Folder do arquivo.

property Cake\Filesystem\File::$**name**

O nome do arquivo com a extensão. É diferente de *File::name()* que retorna o nome sem a extensão.

property Cake\Filesystem\File::$**info**

Uma matriz de informações do arquivo. Ao invés disso use *File::info()*.

property Cake\Filesystem\File::$**handle**

Contém o recurso de manipulador de arquivo se o arquivo for aberto.

property Cake\Filesystem\File::$**lock**

Habilite o bloqueio para leitura e gravação de arquivos.

property Cake\Filesystem\File::$**path**

O caminho absoluto do arquivo atual.

Cake\Filesystem\File::**append**(*string $data*, *boolean $force = false*)

Anexe a string de dados fornecida ao arquivo atual.

Cake\Filesystem\File::**close**()

Fecha o arquivo atual se estiver aberto.

Cake\Filesystem\File::**copy**(*string $dest*, *boolean $overwrite = true*)

Copie o arquivo para o caminho absoluto `$dest`.

Cake\Filesystem\File::**create**()

Cria o arquivo.

Cake\Filesystem\File::**delete**()

Apaga o arquivo;

Cake\Filesystem\File::**executable**()

Returna `true` se o arquivo for executável

Cake\Filesystem\File::**exists**()

Retorna `true` se o arquivo existe.

Cake\Filesystem\File::**ext**()

Retorna a extensão do arquivo.

Cake\Filesystem\File::**Folder**()

Retorna a pasta atual.

Cake\Filesystem\File::**group**()

Retorna o grupo do arquivo, ou `false` em caso de erro.

Cake\Filesystem\File::**info**()

Retorna as informações do arquivo.

Cake\Filesystem\File::**lastAccess**()

Retorna a hora do último acesso.

Cake\Filesystem\File::**lastChange**()

Retorna a hora da última modificação ou `false` em caso de erro.

Cake\Filesystem\File::**md5**(*integer|boolean $maxsize = 5*)

Obtenha o MD5 Checksum do arquivo com a verificação anterior do tamanho do arquivo, ou `false` no caso de um erro.

Cake\Filesystem\File::**name**()

Retorna o nome do arquivo sem extensão.

Cake\Filesystem\File::**offset**(*integer|boolean $offset = false*, *integer $seek = 0*)

Define ou obtém o deslocamento do arquivo aberto no momento.

Cake\Filesystem\File::**open**(*string $mode = 'r'*, *boolean $force = false*)

Abre o arquivo atual com o $mode fornecido.

Cake\Filesystem\File::**owner**()

Retorna o proprietário do arquivo.

Cake\Filesystem\File::**perms**()

Retorna o «chmod» (permissões) do arquivo.

static Cake\Filesystem\File::**prepare**(*string $data*, *boolean $forceWindows = false*)

Prepara uma string ascii para escrita. Converte as terminações de linha no terminador correto para a plataforma atual. Para Windows, será usado «\r\n», para todas as outras plataformas «\ n».

Cake\Filesystem\File::**pwd**()

Retorna o caminho completo do arquivo.

Cake\Filesystem\File::**read**(*string $bytes = false*, *string $mode = 'rb'*, *boolean $force = false*)

Retorne o conteúdo do arquivo atual como uma string ou retorne `false` em caso de falha.

Cake\Filesystem\File::**readable**()

Retorna `true` se o arquivo é legível.

Cake\Filesystem\File::**safe**(*string $name = null*, *string $ext = null*)

Torna o nome do arquivo seguro para salvar.

Cake\Filesystem\File::**size**()

Retorna o tamanho do arquivo em bytes.

Cake\Filesystem\File::**writable**()

Retorna `true` se o arquivo for gravável.

Cake\Filesystem\File::**write**(*string $data*, *string $mode = 'w'*, *boolean$force = false*)

Grave os dados fornecidos no arquivo atual.

Cake\Filesystem\File::**mime**()

Pega o tipo MIME do arquivo, retorna `false` em caso de falha.

Cake\Filesystem\File::**replaceText**(*$search*, *$replace*)

Substitui o texto em um arquivo. Retorna `false` em caso de falha e `true` em caso de sucesso.

CAPÍTULO 33

Hash

class Cake\Utility**Hash**

O gerenciamento de matrizes, se feito da maneira certa, pode ser uma ferramenta muito poderosa e útil para construir um código mais inteligente e otimizado. O CakePHP oferece um conjunto muito útil de utilitários estáticos na classe Hash que permitem que você faça exatamente isso.

A classe Hash do CakePHP pode ser chamada de qualquer template ou controlador da mesma forma que o Inflector é chamado. Exemplo: *Hash::combine()*.

Sintaxe do Caminho de Hash

A sintaxe de caminho descrita abaixo é usada por todos os métodos em `Hash`. Nem todas as partes da sintaxe do caminho estão disponíveis em todos os métodos. Uma expressão de caminho é feita de qualquer número de tokens. Os tokens são compostos por dois grupos. Expressões são usadas para percorrer os dados da matriz, enquanto as expressões são usadas para qualificar elementos.

Tipos de Expressão

Expressão	Definição
`{n}`	Representa uma chave numérica. Irá corresponder a qualquer string ou chave numérica
`{s}`	Representa uma string. Irá corresponder a qualquer valor de string, incluindo valores de string numéricos.
`{*}`	Corresponde a qualquer valor.
`Foo`	Corresponde às chaves exatamente com o mesmo valor.

Todos os elementos de expressão são suportados por todos os métodos. Além de elementos de expressão, você pode usar a correspondência de atributos com certos métodos. Eles são: `extract()`, `combine()`, `format()`, `check()`, `map()`, `reduce()`, `apply()`, `sort()`, `insert()`, `remove()` e `nest()`.

Tipos de Correspondência de Atributos

Expressão	Definição
`[id]`	Combine elementos com uma determinada chave de array.
`[id=2]`	Combine elementos com id igual a 2.
`[id!=2]`	Combine elementos com id diferente de 2.
`[id>2]`	Combine elementos com id maior que 2.
`[id>=2]`	Combine elementos com id maior ou igual a 2.
`[id<2]`	Combine elementos com id menor que 2
`[id<=2]`	Combine elementos com id menor ou igual a 2.
`[text=/.../]`	Combine elementos que possuem valores correspondentes à expressão regular dentro de `...`.

static Cake\Utility\Hash::**get**(*array|\ArrayAccess $data*, *$path*, *$default = null*)

`get()` é uma versão simplificada de `extract()`, ele só suporta expressões de caminho direto. Caminhos como `{n}`, `{s}`, `{*}` ou expressões não são suportados. Use `get()` quando quiser exatamente um valor de uma matriz. Se um caminho correspondente não for encontrado, o valor padrão será retornado.

static Cake\Utility\Hash::**extract**(*array|\ArrayAccess $data*, *$path*)

`Hash::extract()` suporta todas as expressões e componentes de correspondência *Sintaxe do Caminho de Hash*. Você pode usar a extração para recuperar dados de matrizes ou objetos que implementam a interface `ArrayAccess`, ao longo de caminhos arbitrários rapidamente, sem ter que percorrer as estruturas de dados. Em vez disso, você usa expressões de caminho para qualificar quais elementos você deseja que sejam retornados:

```
// Uso comum:
$users = [
    ['id' => 1, 'name' => 'mark'],
    ['id' => 2, 'name' => 'jane'],
    ['id' => 3, 'name' => 'sally'],
    ['id' => 4, 'name' => 'jose'],
];
$results = Hash::extract($users, '{n}.id');
// $results é igual a:
// [1,2,3,4];
```

static Cake\Utility\Hash::**insert**(*array $data*, *$path*, *$values = null*)

Insere `$values` em uma matriz conforme definido por `$path`:

```
$a = [
    'pages' => ['name' => 'page']
];
$result = Hash::insert($a, 'files', ['name' => 'files']);
// $result agora parece:
[
    [pages] => [
        [name] => page
    ]
    [files] => [
```

(continues on next page)

(continuação da página anterior)

```
        [name] => files
    ]
]
```

Você pode usar caminhos usando `{n}`, `{s}` e `{*}` para inserir dados em vários pontos:

```
$users = Hash::insert($users, '{n}.new', 'value');
```

As expressões de atributos funcionam com `insert()` também:

```
$data = [
    0 => ['up' => true, 'Item' => ['id' => 1, 'title' => 'first']],
    1 => ['Item' => ['id' => 2, 'title' => 'second']],
    2 => ['Item' => ['id' => 3, 'title' => 'third']],
    3 => ['up' => true, 'Item' => ['id' => 4, 'title' => 'fourth']],
    4 => ['Item' => ['id' => 5, 'title' => 'fifth']],
];
$result = Hash::insert($data, '{n}[up].Item[id=4].new', 9);
/* $result agora se parece:
    [
        ['up' => true, 'Item' => ['id' => 1, 'title' => 'first']],
        ['Item' => ['id' => 2, 'title' => 'second']],
        ['Item' => ['id' => 3, 'title' => 'third']],
        ['up' => true, 'Item' => ['id' => 4, 'title' => 'fourth', 'new' => 9]],
        ['Item' => ['id' => 5, 'title' => 'fifth']],
    ]
*/
```

static Cake\Utility\Hash::**remove**(*array $data*, *$path*)

Remove todos os elementos de uma matriz que corresponde a `$path`.

```
$a = [
    'pages' => ['name' => 'page'],
    'files' => ['name' => 'files']
];
$result = Hash::remove($a, 'files');
/* $result agora se parece:
    [
        [pages] => [
            [name] => page
        ]

    ]
*/
```

Usando `{n}`, `{s}` e `{*}` permitirá que você remova múltiplos valores de uma vez. Você também pode usar expressões de atributo com `remove()`:

```
$data = [
    0 => ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
    1 => ['Item' => ['id' => 2, 'title' => 'second']],
    2 => ['Item' => ['id' => 3, 'title' => 'third']],
    3 => ['clear' => true, 'Item' => ['id' => 4, 'title' => 'fourth']],
```

(continues on next page)

(continuação da página anterior)

```
    4 => ['Item' => ['id' => 5, 'title' => 'fifth']],
];
$result = Hash::remove($data, '{n}[clear].Item[id=4]');
/* $result agora se parece:
    [
        ['clear' => true, 'Item' => ['id' => 1, 'title' => 'first']],
        ['Item' => ['id' => 2, 'title' => 'second']],
        ['Item' => ['id' => 3, 'title' => 'third']],
        ['clear' => true],
        ['Item' => ['id' => 5, 'title' => 'fifth']],
    ]
*/
```

static Cake\Utility\Hash::**combine**(*array $data*, *$keyPath*, *$valuePath = null*, *$groupPath = null*)

Cria uma matriz associativa usando um `$keyPath` como o caminho para construir suas chaves, e opcionalmente `$valuePath` como o caminho para obter os valores. Se `$valuePath` não for especificado, ou não corresponder a nada, os valores serão inicializados como nulos. Você pode opcionalmente agrupar os valores pelo que é obtido ao seguir o caminho especificado em `$groupPath`.:

```
$a = [
    [
        'User' => [
            'id' => 2,
            'group_id' => 1,
            'Data' => [
                'user' => 'mariano.iglesias',
                'name' => 'Mariano Iglesias'
            ]
        ]
    ],
    [
        'User' => [
            'id' => 14,
            'group_id' => 2,
            'Data' => [
                'user' => 'phpnut',
                'name' => 'Larry E. Masters'
            ]
        ]
    ],
];

$result = Hash::combine($a, '{n}.User.id');
/* $result agora se parece com:
    [
        [2] =>
        [14] =>
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.user');
/* $result agora se parece com:
```

(continues on next page)

(continuação da página anterior)

```
    [
        [2] => 'mariano.iglesias'
        [14] => 'phpnut'
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data');
/* $result agora se parece com:
    [
        [2] => [
                [user] => mariano.iglesias
                [name] => Mariano Iglesias
        ]
        [14] => [
                [user] => phpnut
                [name] => Larry E. Masters
        ]
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name');
/* $result agora se parece com:
    [
        [2] => Mariano Iglesias
        [14] => Larry E. Masters
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data', '{n}.User.group_id');
/* $result agora se parece com:
    [
        [1] => [
                [2] => [
                        [user] => mariano.iglesias
                        [name] => Mariano Iglesias
                ]
        ]
        [2] => [
                [14] => [
                        [user] => phpnut
                        [name] => Larry E. Masters
                ]
        ]
    ]
*/

$result = Hash::combine($a, '{n}.User.id', '{n}.User.Data.name', '{n}.User.group_id
↪');
/* $result agora se parece com:
    [
        [1] => [
                [2] => Mariano Iglesias
```

(continues on next page)

(continuação da página anterior)

```
        ]
        [2] => [
                [14] => Larry E. Masters
        ]
    ]
*/

// A partir de 3.9.0 $keyPath pode ser nulo
$result = Hash::combine($a, null, '{n}.User.Data.name');
/* $result agora se parece com:
    [
        [0] => Mariano Iglesias
        [1] => Larry E. Masters
    ]
*/
```

Você pode fornecer matrizes para `$keyPath` e `$valuePath`. Se você fizer isso, o primeiro valor será usado com o formato de string, para valores extraídos por outros caminhos:

```
$result = Hash::combine(
    $a,
    '{n}.User.id',
    ['%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'],
    '{n}.User.group_id'
);
/* $result agora se parece com:
    [
        [1] => [
                [2] => mariano.iglesias: Mariano Iglesias
        ]
        [2] => [
                [14] => phpnut: Larry E. Masters
        ]
    ]
*/

$result = Hash::combine(
    $a,
    ['%s: %s', '{n}.User.Data.user', '{n}.User.Data.name'],
    '{n}.User.id'
);
/* $result agora se parece com:
    [
        [mariano.iglesias: Mariano Iglesias] => 2
        [phpnut: Larry E. Masters] => 14
    ]
*/
```

static Cake\Utility\Hash::**format**(*array $data*, *array $paths*, *$format*)

Retorna uma série de valores extraídos de uma matriz, formatados com uma string:

```
$data = [
```

(continues on next page)

(continuação da página anterior)

```
    [
        'Person' => [
            'first_name' => 'Nate',
            'last_name' => 'Abele',
            'city' => 'Boston',
            'state' => 'MA',
            'something' => '42'
        ]
    ],
    [
        'Person' => [
            'first_name' => 'Larry',
            'last_name' => 'Masters',
            'city' => 'Boondock',
            'state' => 'TN',
            'something' => '{0}'
        ]
    ],
    [
        'Person' => [
            'first_name' => 'Garrett',
            'last_name' => 'Woodworth',
            'city' => 'Venice Beach',
            'state' => 'CA',
            'something' => '{1}'
        ]
    ]
];

$res = Hash::format($data, ['{n}.Person.first_name', '{n}.Person.something'], '%2$d,
↪ %1$s');
/*
[
    [0] => 42, Nate
    [1] => 0, Larry
    [2] => 0, Garrett
]
*/

$res = Hash::format($data, ['{n}.Person.first_name', '{n}.Person.something'], '%1$s,
↪ %2$d');
/*
[
    [0] => Nate, 42
    [1] => Larry, 0
    [2] => Garrett, 0
]
*/
```

static Cake\Utility\Hash::**contains**(*array $data*, *array $needle*)

Determina se um Hash ou matriz contém as chaves e valores exatos de outro:

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
    2 => ['name' => 'contact'],
    'a' => 'b'
];

$result = Hash::contains($a, $a);
// true
$result = Hash::contains($a, $b);
// false
$result = Hash::contains($b, $a);
// true
```

static Cake\Utility\Hash::**check**(*array $data*, *string $path = null*)

Verifica se um determinado caminho está definido em uma matriz:

```
$set = [
    'My Index 1' => ['First' => 'The first item']
];
$result = Hash::check($set, 'My Index 1.First');
// $result == true

$result = Hash::check($set, 'My Index 1');
// $result == true

$set = [
    'My Index 1' => [
        'First' => [
            'Second' => [
                'Third' => [
                    'Fourth' => 'Heavy. Nesting.'
                ]
            ]
        ]
    ]
];
$result = Hash::check($set, 'My Index 1.First.Second');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Second.Third');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Second.Third.Fourth');
// $result == true

$result = Hash::check($set, 'My Index 1.First.Seconds.Third.Fourth');
// $result == false
```

static Cake\Utility\Hash::**filter**(*array $data, $callback = ['Hash', 'filter']*)

Filtra os elementos vazios da matriz, excluindo "0". Você também pode fornecer um `$callback` personalizado para filtrar os elementos da matriz. O retorno de chamada deve retornar `false` para remover elementos da matriz resultante:

```
$data = [
    '0',
    false,
    true,
    0,
    ['one thing', 'I can tell you', 'is you got to be', false]
];
$res = Hash::filter($data);

/* $res agora se parece:
    [
        [0] => 0
        [2] => true
        [3] => 0
        [4] => [
                [0] => one thing
                [1] => I can tell you
                [2] => is you got to be
        ]
    ]
*/
```

static Cake\Utility\Hash::**flatten**(*array $data*, *string $separator = '.'*)

Nivela uma matriz multidimensional em uma única dimensão:

```
$arr = [
    [
        'Post' => ['id' => '1', 'title' => 'First Post'],
        'Author' => ['id' => '1', 'user' => 'Kyle'],
    ],
    [
        'Post' => ['id' => '2', 'title' => 'Second Post'],
        'Author' => ['id' => '3', 'user' => 'Crystal'],
    ],
];
$res = Hash::flatten($arr);
/* $res now looks like:
    [
        [0.Post.id] => 1
        [0.Post.title] => First Post
        [0.Author.id] => 1
        [0.Author.user] => Kyle
        [1.Post.id] => 2
        [1.Post.title] => Second Post
        [1.Author.id] => 3
        [1.Author.user] => Crystal
    ]
*/
```

static Cake\Utility\Hash::**expand**(*array $data*, *string $separator = '.'*)

Expande uma matriz que foi previamente achatada com *Hash::flatten()*:

```
$data = [
    '0.Post.id' => 1,
    '0.Post.title' => First Post,
    '0.Author.id' => 1,
    '0.Author.user' => Kyle,
    '1.Post.id' => 2,
    '1.Post.title' => Second Post,
    '1.Author.id' => 3,
    '1.Author.user' => Crystal,
];
$res = Hash::expand($data);
/* $res agora se parece com:
[
    [
        'Post' => ['id' => '1', 'title' => 'First Post'],
        'Author' => ['id' => '1', 'user' => 'Kyle'],
    ],
    [
        'Post' => ['id' => '2', 'title' => 'Second Post'],
        'Author' => ['id' => '3', 'user' => 'Crystal'],
    ],
];
*/
```

static Cake\Utility\Hash::**merge**(*array $data*, *array $merge*[, *array $n*])

Esta função pode ser considerada um híbrido entre `array_merge` e `array_merge_recursive` do PHP. A diferença entre as duas é que se uma chave da matriz contém outra matriz, então a função se comporta recursivamente (ao contrário de `array_merge`), mas não se comporta do mesmo jeito para chaves contendo strings (ao contrário de `array_merge_recursive`).

Nota: Esta função funcionará com uma quantidade ilimitada de argumentos e casting de parâmetros primitivos para matrizes.

```
$array = [
    [
        'id' => '48c2570e-dfa8-4c32-a35e-0d71cbdd56cb',
        'name' => 'mysql raleigh-workshop-08 < 2008-09-05.sql ',
        'description' => 'Importing an sql dump'
    ],
    [
        'id' => '48c257a8-cf7c-4af2-ac2f-114ecbdd56cb',
        'name' => 'pbpaste | grep -i Unpaid | pbcopy',
        'description' => 'Remove all lines that say "Unpaid".',
    ]
];
$arrayB = 4;
$arrayC = [0 => "test array", "cats" => "dogs", "people" => 1267];
$arrayD = ["cats" => "felines", "dog" => "angry"];
$res = Hash::merge($array, $arrayB, $arrayC, $arrayD);
```

(continues on next page)

(continuação da página anterior)

```
/* $res agora se parece com:
[
    [0] => [
            [id] => 48c2570e-dfa8-4c32-a35e-0d71cbdd56cb
            [name] => mysql raleigh-workshop-08 < 2008-09-05.sql
            [description] => Importing an sql dump
    ]
    [1] => [
            [id] => 48c257a8-cf7c-4af2-ac2f-114ecbdd56cb
            [name] => pbpaste | grep -i Unpaid | pbcopy
            [description] => Remove all lines that say "Unpaid".
    ]
    [2] => 4
    [3] => test array
    [cats] => felines
    [people] => 1267
    [dog] => angry
]
*/
```

static Cake\Utility\Hash::**numeric**(*array $data*)

Verifica se todos os valores da matriz são numéricas:

```
$data = ['one'];
$res = Hash::numeric(array_keys($data));
// $res é true

$data = [1 => 'one'];
$res = Hash::numeric($data);
// $res é false
```

static Cake\Utility\Hash::**dimensions**(*array $data*)

Conta as dimensões de uma matriz. Este método irá considerar apenas a dimensão do primeiro elemento na matriz:

```
$data = ['one', '2', 'three'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => '1.1', '2', '3'];
$result = Hash::dimensions($data);
// $result == 1

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 2

$data = ['1' => '1.1', '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::dimensions($data);
// $result == 1
```

(continues on next page)

(continuação da página anterior)

```
$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => ['3.1.1' => '3.1.1.1']]];
$result = Hash::dimensions($data);
// $result == 2
```

static Cake\Utility\Hash::**maxDimensions**(*array $data*)

Semelhante a *dimensions()*, no entanto, este método retorna, o maior número de dimensões de qualquer elemento na matriz:

```
$data = ['1' => '1.1', '2', '3' => ['3.1' => '3.1.1']];
$result = Hash::maxDimensions($data);
// $result == 2

$data = ['1' => ['1.1' => '1.1.1'], '2', '3' => ['3.1' => ['3.1.1' => '3.1.1.1']]];
$result = Hash::maxDimensions($data);
// $result == 3
```

static Cake\Utility\Hash::**map**(*array $data*, *$path*, *$function*)

Cria uma nova matriz, extraindo `$path`, e mapeando `$function` nos resultados. Você pode usar expressões e elementos correspondentes com este método:

```
// Chame a função noop $this->noop() em cada elemento de $data
$result = Hash::map($data, "{n}", [$this, 'noop']);

public function noop(array $array)
{
    // Faça coisas para a matriz e retorne o resultado
    return $array;
}
```

static Cake\Utility\Hash::**reduce**(*array $data*, *$path*, *$function*)

Cria um único valor, extraindo `$path`, e reduzindo os resultados extraídos com `$function`. Você pode usar expressões e elementos correspondentes com este método.

static Cake\Utility\Hash::**apply**(*array $data*, *$path*, *$function*)

Aplique um retorno de chamada a um conjunto de valores extraídos usando `$function`. A função obterá os valores extraídos do primeiro argumento:

```
$data = [
    ['date' => '01-01-2016', 'booked' => true],
    ['date' => '01-01-2016', 'booked' => false],
    ['date' => '02-01-2016', 'booked' => true]
];
$result = Hash::apply($data, '{n}[booked=true].date', 'array_count_values');
/* $result agora parece com:
    [
        '01-01-2016' => 1,
        '02-01-2016' => 1,
    ]
*/
```

static Cake\Utility\Hash::**sort**(*array $data*, *$path*, *$dir*, *$type = 'regular'*)

Classifica uma matriz por qualquer valor, determinado por *Sintaxe do Caminho de Hash* Somente elementos de expressão são suportados por este método:

```
$a = [
    0 => ['Person' => ['name' => 'Jeff']],
    1 => ['Shirt' => ['color' => 'black']]
];
$result = Hash::sort($a, '{n}.Person.name', 'asc');
/* $result agora parece com:
    [
        [0] => [
                [Shirt] => [
                        [color] => black
                ]
        ]
        [1] => [
                [Person] => [
                        [name] => Jeff
                ]
        ]
    ]
*/
```

`$dir` pode ser `asc` ou `desc`. `$type` pode ser um dos seguintes valores:

- `regular` para ordenamento padrão
- `numeric` para classificar valores como seus equivalentes numéricos.
- `string` para classificar valores como seu valor de string.
- `natural` para classificar valores de uma forma amigável ao humano. Classificará `foo10` abaixo de `foo2` por exemplo.

static Cake\Utility\Hash::**diff**(*array $data*, *array $compare*)

Calcula a diferença entre duas matrizes:

```
$a = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about']
];
$b = [
    0 => ['name' => 'main'],
    1 => ['name' => 'about'],
    2 => ['name' => 'contact']
];

$result = Hash::diff($a, $b);
/* $result agora parece com:
    [
        [2] => [
                [name] => contact
        ]
    ]
*/
```

static Cake\Utility\Hash::**mergeDiff**(*array $data*, *array $compare*)

Essa função mescla duas matrizes e empurra as diferenças nos dados para a parte inferior da matriz resultante.

Exemplo 1

```
$array1 = ['ModelOne' => ['id' => 1001, 'field_one' => 'a1.m1.f1', 'field_two' =>
↪'a1.m1.f2']];
$array2 = ['ModelOne' => ['id' => 1003, 'field_one' => 'a3.m1.f1', 'field_two' =>
↪'a3.m1.f2', 'field_three' => 'a3.m1.f3']];
$res = Hash::mergeDiff($array1, $array2);

/* $res agora parece com:
    [
        [ModelOne] => [
                [id] => 1001
                [field_one] => a1.m1.f1
                [field_two] => a1.m1.f2
                [field_three] => a3.m1.f3
            ]
    ]
*/
```

Exemplo 2

```
$array1 = ["a" => "b", 1 => 20938, "c" => "string"];
$array2 = ["b" => "b", 3 => 238, "c" => "string", ["extra_field"]];
$res = Hash::mergeDiff($array1, $array2);
/* $res agora parece com:
    [
        [a] => b
        [1] => 20938
        [c] => string
        [b] => b
        [3] => 238
        [4] => [
                [0] => extra_field
        ]
    ]
*/
```

`static` Cake\Utility\Hash::**normalize**(*array $data*, *$assoc = true*)

Normaliza uma matriz. Se `$assoc` for `true`, a matriz resultante será normalizada para ser uma matriz associativa. Chaves numéricas com valores serão convertidas em chaves de string com valores nulos. Normalizar uma matriz torna o uso dos resultados com *Hash::merge()* mais fácil:

```
$a = ['Tree', 'CounterCache',
    'Upload' => [
        'folder' => 'products',
        'fields' => ['image_1_id', 'image_2_id']
    ]
];
$result = Hash::normalize($a);
/* $result agora parece com:
    [
        [Tree] => null
        [CounterCache] => null
        [Upload] => [
```

(continues on next page)

(continuação da página anterior)

```
                [folder] => products
                [fields] => [
                        [0] => image_1_id
                        [1] => image_2_id
                ]
        ]
    ]
*/

$b = [
    'Cacheable' => ['enabled' => false],
    'Limit',
    'Bindable',
    'Validator',
    'Transactional'
];
$result = Hash::normalize($b);
/* $result agora parece com:
    [
        [Cacheable] => [
                [enabled] => false
        ]

        [Limit] => null
        [Bindable] => null
        [Validator] => null
        [Transactional] => null
    ]
*/
```

static Cake\Utility\Hash::**nest**(*array $data*, *array $options = []*)

Pega um conjunto de matriz simples e cria uma estrutura de dados aninhada ou encadeada.

Opções:

- `children` O nome da chave a ser usada no conjunto de resultados para os valores aninhados. O padrão é "children".
- `idPath` O caminho para uma chave que identifica cada entrada. Deve ser compatível com *Hash::extract()*. O padrão é `{n}.$alias.id`
- `parentPath` O caminho para uma chave que identifica o pai de cada entrada. Deve ser compatível com *Hash::extract()*. O padrão é `{n}.$alias.parent_id`
- `root` O id do resultado desejado mais alto.

Por exemplo, se você tivesse a seguinte matriz de dados:

```
$data = [
    ['ThreadPost' => ['id' => 1, 'parent_id' => null]],
    ['ThreadPost' => ['id' => 2, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 3, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 4, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 5, 'parent_id' => 1]],
    ['ThreadPost' => ['id' => 6, 'parent_id' => null]],
```

(continues on next page)

(continuação da página anterior)

```
    ['ThreadPost' => ['id' => 7, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 8, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 9, 'parent_id' => 6]],
    ['ThreadPost' => ['id' => 10, 'parent_id' => 6]]
];

$result = Hash::nest($data, ['root' => 6]);
/* $result agora parece com:
    [
        (int) 0 => [
            'ThreadPost' => [
                'id' => (int) 6,
                'parent_id' => null
            ],
            'children' => [
                (int) 0 => [
                    'ThreadPost' => [
                        'id' => (int) 7,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 1 => [
                    'ThreadPost' => [
                        'id' => (int) 8,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 2 => [
                    'ThreadPost' => [
                        'id' => (int) 9,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ],
                (int) 3 => [
                    'ThreadPost' => [
                        'id' => (int) 10,
                        'parent_id' => (int) 6
                    ],
                    'children' => []
                ]
            ]
        ]
    ]
    */
```

CAPÍTULO 34

Cliente Http

class Cake\Http**Client**(*mixed $config = []*)

O CakePHP inclui um cliente HTTP compatível com a PSR-18 que pode ser usado para fazer solicitações. É uma ótima maneira de se comunicar com serviços da web e APIs remotas.

Fazendo Solicitações

Fazer solicitações é simples e direto. Fazer uma solicitação GET parece:

```
use Cake\Http\Client;

$http = new Client();

// GET simples
$response = $http->get('http://example.com/test.html');

// GET simples com querystring
$response = $http->get('http://example.com/search', ['q' => 'widget']);

// GET simples com querystring & cabeçalhos adicionais
$response = $http->get('http://example.com/search', ['q' => 'widget'], [
  'headers' => ['X-Requested-With' => 'XMLHttpRequest']
]);
```

Fazer solicitações POST e PUT é igualmente simples:

```
// Envie uma solicitação POST com dados codificados em application/x-www-form-urlencoded
$http = new Client();
```

(continues on next page)

(continuação da página anterior)

```
$response = $http->post('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post'
]);

// Envie uma solicitação PUT com dados codificados application/x-www-form-urlencoded
$response = $http->put('http://example.com/posts/add', [
  'title' => 'testing',
  'body' => 'content in the post'
]);

// Outros métodos também.
$http->delete(/* ... */);
$http->head(/* ... */);
$http->patch(/* ... */);
```

Se você criou um objeto de solicitação PSR-7, pode enviá-lo usando `sendRequest()`:

```
use Cake\Http\Client;
use Cake\Http\Client\Request as ClientRequest;

$request = new ClientRequest(
    'http://example.com/search',
    ClientRequest::METHOD_GET
);
$client = new Client();
$response = $client->sendRequest($request);
```

Criação de Solicitações Multipart com Arquivos

Você pode incluir arquivos em corpos de solicitação:

```
$http = new Client();
$response = $http->post('http://example.com/api', [
  'image' => fopen('/path/to/a/file', 'r'),
]);
```

O arquivo será lido até o fim; não será rebobinado antes de ser lido.

Criação de Corpos de Solicitação de Várias Partes Manualmente

Pode haver momentos em que você precise criar um corpo de solicitação de uma maneira muito específica. Nessas situações, você pode frequentemente usar `Cake\Http\Client\FormData` para criar a solicitação HTTP multipart que você deseja:

```
use Cake\Http\Client\FormData;

$data = new FormData();

// Crie uma parte XML
```

(continues on next page)

(continuação da página anterior)

```
$xml = $data->newPart('xml', $xmlString);
// Defina o tipo de conteúdo.
$xml->type('application/xml');
$data->add($xml);

// Crie um upload de arquivo com addFile()
// Isso irá anexar o arquivo aos dados do formulário também.
$file = $data->addFile('upload', fopen('/some/file.txt', 'r'));
$file->contentId('abc123');
$file->disposition('attachment');

// Envie a solicitação.
$response = $http->post(
    'http://example.com/api',
    (string)$data,
    ['headers' => ['Content-Type' => $data->contentType()]]
);
```

Enviando o Corpo da Solicitação

Ao lidar com APIs REST, você geralmente precisa enviar corpos de solicitação que não são codificados por formulário. Http\Cliente expõe isso através da opção de tipo:

```
// Envie um corpo de solicitação JSON.
$http = new Client();
$response = $http->post(
  'http://example.com/tasks',
  json_encode($data),
  ['type' => 'json']
);
```

A chave `type` pode ser “json”, “xml” ou um tipo MIME completo. Ao usar a opção `type`, você deve fornecer os dados como uma string. Se você estiver fazendo uma solicitação GET que precisa de parâmetros de string de consulta e um corpo de solicitação, você pode fazer o seguinte:

```
// Envie um corpo JSON em uma solicitação GET com parâmetros de string de consulta.
$http = new Client();
$response = $http->get(
  'http://example.com/tasks',
  ['q' => 'test', '_content' => json_encode($data)],
  ['type' => 'json']
);
```

Opções de Método para Solicitação

Cada método HTTP leva um parâmetro `$options` que é usado para fornecer informações adicionais de solicitação. As seguintes chaves podem ser usadas em `$options`:

- `headers` - Matriz de cabeçalhos adicionais
- `cookie` - Matriz de cookies para usar.
- `proxy` - Matriz de informações do proxy.
- `auth` - Matriz de dados de autenticação, a chave `type` é usada para delegar a uma estratégia de autenticação. Por padrão, a autenticação básica é usada.
- `ssl_verify_peer` - o padrão é `true`. Defina como `false` para desativar a verificação de certificação SSL (não recomendado).
- `ssl_verify_peer_name` - o padrão é `true`. Defina como `false` para desabilitar a verificação do nome do host ao verificar os certificados SSL (não recomendado).
- `ssl_verify_depth` - o padrão é 5. Profundidade a ser percorrida na cadeia de CA.
- `ssl_verify_host` - o padrão é `true`. Valide o certificado SSL em relação ao nome do host.
- `ssl_cafile` - o padrão é construído em cafile. Substitua para usar pacotes CA personalizados.
- `timeout` - Duração de espera antes de expirar em segundos.
- `type` - Envie um corpo de solicitação em um tipo de conteúdo personalizado. Requer que `$data` seja uma string ou que a opção `_content` seja definida ao fazer solicitações GET.
- `redirect` - Número de redirecionamentos a seguir. O padrão é `false`.

O parâmetro options é sempre o terceiro parâmetro em cada um dos métodos HTTP. Eles também podem ser usados ao construir `Client` para criar *scoped clients*.

Autenticação

`Cake\Http\Client` suporta alguns sistemas de autenticação. Diferentes estratégias de autenticação podem ser adicionadas pelos desenvolvedores. As estratégias de autenticação são chamadas antes do envio da solicitação e permitem que cabeçalhos sejam adicionados ao contexto da solicitação.

Usando Autenticação Básica

Um exemplo de autenticação básica:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => ['username' => 'mark', 'password' => 'secret']
]);
```

Por padrão, o `Cake\Http\Client` usará a autenticação básica se não houver uma chave `'type'` na opção auth.

Usando a Autenticação Digest

Um exemplo de autenticação básica:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => [
    'type' => 'digest',
    'username' => 'mark',
    'password' => 'secret',
    'realm' => 'myrealm',
    'nonce' => 'onetimevalue',
    'qop' => 1,
    'opaque' => 'someval'
  ]
]);
```

Ao definir a chave "type" como "digest", você informa ao subsistema de autenticação para usar a autenticação digest.

Autenticação OAuth 1

Muitos serviços da web moderna exigem autenticação OAuth para acessar suas APIs. A autenticação OAuth incluída pressupõe que você já tenha sua chave e segredo do consumidor:

```
$http = new Client();
$response = $http->get('http://example.com/profile/1', [], [
  'auth' => [
    'type' => 'oauth',
    'consumerKey' => 'bigkey',
    'consumerSecret' => 'secret',
    'token' => '...',
    'tokenSecret' => '...',
    'realm' => 'tickets',
  ]
]);
```

Autenticação OAuth 2

Como OAuth2 geralmente é um único cabeçalho, não há um adaptador de autenticação especializado. Em vez disso, você pode criar um cliente com o token de acesso:

```
$http = new Client([
    'headers' => ['Authorization' => 'Bearer ' . $accessToken]
]);
$response = $http->get('https://example.com/api/profile/1');
```

Autenticação no Proxy

Alguns proxies requerem autenticação para serem usados. Geralmente, essa autenticação é Básica, mas pode ser implementada por qualquer adaptador de autenticação. Por padrão, o Http\Client assumirá a autenticação Básica, a menos que a chave de tipo seja definida:

```
$http = new Client();
$response = $http->get('http://example.com/test.php', [], [
  'proxy' => [
    'username' => 'mark',
    'password' => 'testing',
    'proxy' => '127.0.0.1:8080',
  ]
]);
```

O segundo parâmetro de proxy deve ser uma string com um IP ou um domínio sem protocolo. As informações de nome de usuário e senha serão passadas pelos cabeçalhos da solicitação, enquanto a string do proxy será passada por stream_context_create()[136].

Criação de Clientes com Escopo

Ter que redigitar o nome de domínio, as configurações de autenticação e proxy pode se tornar tedioso e sujeito a erros. Para reduzir a chance de erro e aliviar um pouco do tédio, você pode criar clientes com escopo:

```
// Crie um cliente com escopo definido.
$http = new Client([
  'host' => 'api.example.com',
  'scheme' => 'https',
  'auth' => ['username' => 'mark', 'password' => 'testing']
]);

// Faça uma solicitação para api.example.com
$response = $http->get('/test.php');
```

As seguintes informações podem ser usadas ao criar um cliente com escopo:

- host
- scheme
- proxy
- auth
- port
- cookies
- timeout
- ssl_verify_peer
- ssl_verify_depth
- ssl_verify_host

[136] https://php.net/manual/en/function.stream-context-create.php

Qualquer uma dessas opções pode ser substituída, especificando-as ao fazer solicitações. host, scheme, proxy, port são substituídos no URL do pedido:

```
// Usando o cliente com escopo criado anteriormente.
$response = $http->get('http://foo.com/test.php');
```

O exemplo acima irá substituir o domínio, esquema e porta. No entanto, essa solicitação continuará usando todas as outras opções definidas quando o cliente com escopo foi criado. Veja *Opções de Método para Solicitação* para mais informações sobre as opções suportadas.

Configuração e Gerenciamento de Cookies

Http\Client também pode aceitar cookies ao fazer solicitações. Além de aceitar cookies, ele também armazenará automaticamente cookies válidos definidos nas respostas. Qualquer resposta com cookies, os terá armazenados na instância de origem do Http\Client. Os cookies armazenados em uma instância do cliente são incluídos automaticamente em solicitações futuras para combinações de domínio + caminho que corresponderem:

```
$http = new Client([
    'host' => 'cakephp.org'
]);

// Faça uma solicitação que defina alguns cookies
$response = $http->get('/');

// Os cookies da primeira solicitação serão incluídos
// por padrão.
$response2 = $http->get('/changelogs');
```

Você sempre pode substituir os cookies incluídos automaticamente, definindo-os nos parâmetros `$options` da solicitação:

```
// Substitua um cookie armazenado por um valor personalizado.
$response = $http->get('/changelogs', [], [
    'cookies' => ['sessionid' => '123abc']
]);
```

Você pode adicionar objetos de cookie ao cliente após criá-lo usando o método `addCookie()`:

```
use Cake\Http\Cookie\Cookie;

$http = new Client([
    'host' => 'cakephp.org'
]);
$http->addCookie(new Cookie('session', 'abc123'));
```

Objetos de Resposta

class Cake\Http\Client**Response**

Os objetos de resposta têm vários métodos para inspecionar os dados recebidos.

Leitura do Corpo da Resposta

Você lê todo o corpo da resposta como uma string:

```
// Leia toda a resposta como uma string.
$response->getStringBody();
```

Você também pode acessar o objeto stream para a resposta e usar seus métodos:

```
// Obtêm um Psr\Http\Message\StreamInterface contendo o corpo da resposta
$stream = $response->getBody();

// Leia um fluxo de 100 bytes por vez.
while (!$stream->eof()) {
    echo $stream->read(100);
}
```

Lendo Corpo de Respostas JSON e XML

Como as respostas JSON e XML são comumente usadas, os objetos de resposta fornecem acessores fáceis de usar para ler dados decodificados. Os dados JSON são decodificados em uma matriz, enquanto os dados XML são decodificados em uma árvore `SimpleXMLElement`:

```
// Obtêm algum XML
$http = new Client();
$response = $http->get('http://example.com/test.xml');
$xml = $response->getXml();

// Obtêm algum JSON
$http = new Client();
$response = $http->get('http://example.com/test.json');
$json = $response->getJson();
```

Os dados de resposta decodificados são armazenados no objeto de resposta, portanto, acessá-lo várias vezes não tem custo adicional.

Acessando Cabeçalhos da Resposta

Você pode acessar os cabeçalhos por meio de alguns métodos diferentes. Os nomes dos cabeçalhos são sempre tratados como valores que não diferenciam maiúsculas de minúsculas ao acessá-los por meio de métodos:

```
// Obtenha todos os cabeçalhos como uma matriz associativa.
$response->getHeaders();

// Obtenha um único cabeçalho como uma matriz.
$response->getHeader('content-type');

// Obtenha um cabeçalho como uma string
$response->getHeaderLine('content-type');

// Obtenha a codificação da resposta
$response->getEncoding();
```

Acessando Dados do Cookie

Você pode ler os cookies com alguns métodos diferentes, dependendo de quantos dados você precisa sobre os cookies:

```
// Obtenha todos os cookies (dados completos)
$response->getCookies();

// Obtenha o valor de um único cookie.
$response->getCookie('session_id');

// Obtenha os dados completos para um único cookie,
// incluindo valor, expiração, caminho, httponly, chaves seguras.
$response->getCookieData('session_id');
```

Verificando o Código de Status

Os objetos de resposta fornecem alguns métodos para verificar os códigos de status:

```
// A resposta foi 20x
$response->isOk();

// A resposta foi 30x
$response->isRedirect();

// Obtenha o código de status
$response->getStatusCode();
```

Alteração de Adaptadores de Transporte

Por padrão, o `Http\Client` irá preferir usar um adaptador de transporte baseado em `curl`. Se a extensão curl não estiver disponível, um adaptador baseado em fluxo será usado. Você pode forçar a seleção de um adaptador de transporte usando uma opção de construtor:

```
use Cake\Http\Client\Adapter\Stream;

$client = new Client(['adapter' => Stream::class]);
```

CAPÍTULO 35

Inflector

class Cake\Utility**Inflector**

A classe `Inflector` recebe uma string e a manipula afim de suportar variações de palavas como pluralizações ou CamelCase e normalmente é acessada estaticamente. Exemplo: `Inflector::pluralize('example')` retorna «examples».

Você pode testar as inflexões em inflector.cakephp.org[137].

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.

[137] https://inflector.cakephp.org/

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

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 involvem 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 inflector 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 «lápis», 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 transliterações 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.

CAPÍTULO 36

Número

class Cake\I18n**Number**

Se você precisa das funcionalidades do `NumberHelper` fora da `View`, use a classe `Number`

```
namespace App\Controller;

use Cake\I18n\Number;

class UsersController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Auth');
    }

    public function afterLogin()
    {
        $storageUsed = $this->Auth->user('storage_used');
        if ($storageUsed > 5000000) {
            // Notify users of quota
            $this->Flash->success(__('You are using {0} storage', Number::toReadableSize(
↪$storageUsed)));
        }
    }
}
```

Todas essas funções retornam o número formatado; eles não são impressas automaticamente na visualização de saída.

Formatação de Valores Monetários

`Cake\I18n\Number::currency`(*mixed $value*, *string $currency = null*, *array $options = []*)

Este método é usado para exibir um número em formatos monetários comuns (EUR, GBP, USD), com base no código de moeda ISO 4217 de 3 letras. O uso em uma view se parece com:

```
// Called as NumberHelper
echo $this->Number->currency($value, $currency);

// Called as Number
echo Number::currency($value, $currency);
```

O primeiro parâmetro, `$value`, deve ser um número de ponto flutuante que representa a quantidade de dinheiro que você está expressando. O segundo parâmetro é uma string usada para escolher um esquema de formatação de monetária predefinida:

$ moeda | 1234.56, formatação pelo tipo monetário	
EUR	€1.234,56
GBP	£1,234.56
USD	$1,234.56

O terceiro parâmetro é um conjunto de opções para definir melhor a saída. As seguintes opções estão disponíveis:

Opção	Descrição
before	Texto a ser exibido antes do número renderizado.
after	Texto a ser exibido após o número renderizado.
zero	O texto a ser usado para valores zero; pode ser uma string ou um número. ou seja, 0, “Grátis!”.
places	Número de casas decimais a serem usadas, ou seja, 2
precision	Número máximo de casas decimais a serem usadas, ou seja, 2
locale	O nome do local a ser usado para formatar o número, ou seja. “Fr_FR”.
fractionSymbol	String a ser usada para números fracionários, ou seja, “centavos”.
fractionPosition	Ou “antes” ou “depois” para colocar o símbolo de fração.
pattern	Um padrão de número ICU a ser usado para formatar o número, ou seja. #, ###. 00
useIntlCode	Defina como `true` para substituir o símbolo da moeda pelo código de moeda internacional

Se de `$currency` for `null`, a moeda padrão será retornada em `Cake\I18n\Number::defaultCurrency()`. Para formatar moedas em um formato de contabilidade, você deve definir o formato da moeda:

```
Number::setDefaultCurrencyFormat(Number::FORMAT_CURRENCY_ACCOUNTING);
```

Definição da moeda padrão

`Cake\I18n\Number::setDefaultCurrency`(*$currency*)

Atribui a moeda padrão. Isso elimina a necessidade de sempre passar a moeda para *Cake\I18n\Number::currency()* e alterar todas as saídas de moeda definindo outro padrão. Se `$currency` atribuído o valor `null`, ele apagará o valor armazenado no momento.

Obtendo a moeda padrão

`Cake\I18n\Number::getDefaultCurrency`()

Obtem a moeda padrão. Se a moeda padrão foi definida anteriormente usando `setDefaultCurrency()`, então esse valor será retornado. Por padrão, ele irá retornar o valor `intl.default_locale` do ini se estiver atribuído e `'en_US'` se não estiver.

Formatando números de ponto flutuante

`Cake\I18n\Number::precision`(*float $value*, *int $precision = 3*, *array $options = []*)

Este método exibe um número com a quantidade especificada de precisão (casas decimais). Ele será arredondado para manter o nível de precisão definido.

```
// Called as NumberHelper
echo $this->Number->precision(456.91873645, 2);

// Outputs
456.92

// Called as Number
echo Number::precision(456.91873645, 2);
```

Formatação de Porcentagens

`Cake\I18n\Number::toPercentage`(*mixed $value*, *int $precision = 2*, *array $options = []*)

Opção	Descrição
multiply	Booleano para indicar se o valor deve ser multiplicado por 100. Útil para porcentagens decimais.

Da mesma forma *Cake\I18n\Number::precision()*, ste método formata um número de acordo com a precisão fornecida (onde os números são arredondados para atender à precisão fornecida). Este método também expressa o número como uma porcentagem e anexa a saída com um sinal de porcentagem.

```
// Called as NumberHelper. Output: 45.69%
echo $this->Number->toPercentage(45.691873645);
```

(continues on next page)

(continuação da página anterior)

```
// Called as Number. Output: 45.69%
echo Number::toPercentage(45.691873645);

// Called with multiply. Output: 45.7%
echo Number::toPercentage(0.45691, 1, [
    'multiply' => true
]);
```

Interagindo com valores legíveis para humanos

`Cake\I18n\Number::toReadableSize`(*string $size*)

Este método formata o tamanho dos dados em formatos legíveis por humanos. Ele fornece uma forma de atalho para converter bytes em KB, MB, GB e TB. O tamanho é exibido com um nível de precisão de dois dígitos, de acordo com o tamanho dos dados fornecidos (ou seja, tamanhos maiores são expressos em termos maiores)

```
// Called as NumberHelper
echo $this->Number->toReadableSize(0); // 0 Byte
echo $this->Number->toReadableSize(1024); // 1 KB
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5 GB

// Called as Number
echo Number::toReadableSize(0); // 0 Byte
echo Number::toReadableSize(1024); // 1 KB
echo Number::toReadableSize(1321205.76); // 1.26 MB
echo Number::toReadableSize(5368709120); // 5 GB
```

Formatando Números

`Cake\I18n\Number::format`(*mixed $value*, *array $options = []*)

Este método fornece muito mais controle sobre a formatação de números para uso em suas visualizações (e é usado como o método principal pela maioria dos outros métodos NumberHelper). Usar este método pode ser parecido com:

```
// Called as NumberHelper
$this->Number->format($value, $options);

// Called as Number
Number::format($value, $options);
```

O parâmetro `$value` é o número que você está planejando formatar para saída. Sem o formatting for output. Sem o `$options`, o número 1236.334 produzirá a saída 1,236. Observe que a precisão padrão é zero casas decimais.

O parâmetro `$options` é onde reside a verdadeira magia desse método.

- Se você passar um número inteiro, isso se tornará a quantidade de precisão ou casas para a função.
- Se você passar uma matriz associada, você pode usar as seguintes chaves:

Opção	Descrição
places	Número de casas decimais a serem usadas, ou seja, 2
precision	Número máximo de casas decimais a serem usadas, ou seja, 2
pattern	Um padrão de número ICU a ser usado para formatar o número, ou seja. #, ###. 00
locale	O nome do local a ser usado para formatar o número, ou seja. “Fr_FR”.
before	Texto a ser exibido antes do número renderizado.
after	Texto a ser exibido após o número renderizado.

Exemplo:

```
// Called as NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'

// Called as Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Output '¥ 123,456.79 !'

echo Number::format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Output '123 456,79 !'
```

Cake\I18n\Number::**ordinal**(*mixed $value*, *array $options = []*)

Este método irá gerar um número ordinal.

Exemplos:

```
echo Number::ordinal(1);
// Output '1st'

echo Number::ordinal(2);
// Output '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Output '2e'
```

(continues on next page)

(continuação da página anterior)

```
echo Number::ordinal(410);
// Output '410th'
```

Diferenças de formato

Cake\I18n\Number::**formatDelta**(*mixed $value*, *array $options = []*)

Este método exibe diferenças de valor como um número assinado:

```
// Called as NumberHelper
$this->Number->formatDelta($value, $options);

// Called as Number
Number::formatDelta($value, $options);
```

O parâmetro `$value` é o número que você está planejando formatar para saída. Sem o `$options` , 1236.334 produziria como saída 1,236. Observe que a precisão padrão é zero casas decimais.

O parâmetro `$options` usa as mesmas chaves que *Number::format()*:

Opção	Descrição
places	Número de casas decimais a serem usadas, ou seja, 2
precision	Número máximo de casas decimais a serem usadas, ou seja, 2
locale	O nome do local a ser usado para formatar o número, ou seja. “Fr_FR”.
before	Texto a ser exibido antes do número renderizado.
after	Texto a ser exibido após o número renderizado.

Exemplo:

```
// Called as NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'

// Called as Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Output '[+123,456.79]'
```

Configurar formatadores

Cake\I18n\Number::**config**(*string $locale*, *int $type = NumberFormatter::DECIMAL*, *array $options = []*)

Este método permite configurar padrões do formatador que persistem nas chamadas para vários métodos.

Exemplo:

```
Number::config('en_IN', \NumberFormatter::CURRENCY, [
    'pattern' => '#,##,##0'
]);
```

CAPÍTULO 37

Objetos de Registro

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[138] 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.

[138] https://github.com/cakephp/docs

CAPÍTULO 38

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[139] 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.

[139] https://github.com/cakephp/docs

CAPÍTULO 39

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[140] 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.

[140] https://github.com/cakephp/docs

CAPÍTULO 40

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[141] 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.

[141] https://github.com/cakephp/docs

CAPÍTULO 41

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 corrreta 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 corrreta 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 corrreta 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 corrreta 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-barra 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.

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

CAPÍTULO 42

Debug Kit

Esta página foi movida[142].

[142] https://book.cakephp.org/debugkit/3.x/pt/

CAPÍTULO 43

Migrations

Esta página foi movida[143].

[143] https://book.cakephp.org/migrations/3/pt/

CAPÍTULO 44

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 4.x

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[144] e discutir idéias, o ultimo código e fazer comentários gerais. Se algo mais formal tem que ser proposto ou exite 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 apartir 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 ultimas correções e funcionalidades. Eles são nomeados pela versão a qual se destinam, ex: *3.next*. Uma vez que estas braches estão estáveis elas são fundidas na branch principal da versão.

[144] https://github.com/cakephp?tab=members

- **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.

E sempre, se você tiver alguma questão ou cometários, 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 seqüência de conexão que é formado como um URI. O CakePHP suporta DSN para conexões Cache, Database, Log e Email.

PHP Namespace Index

C

Índice

Símbolos

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z