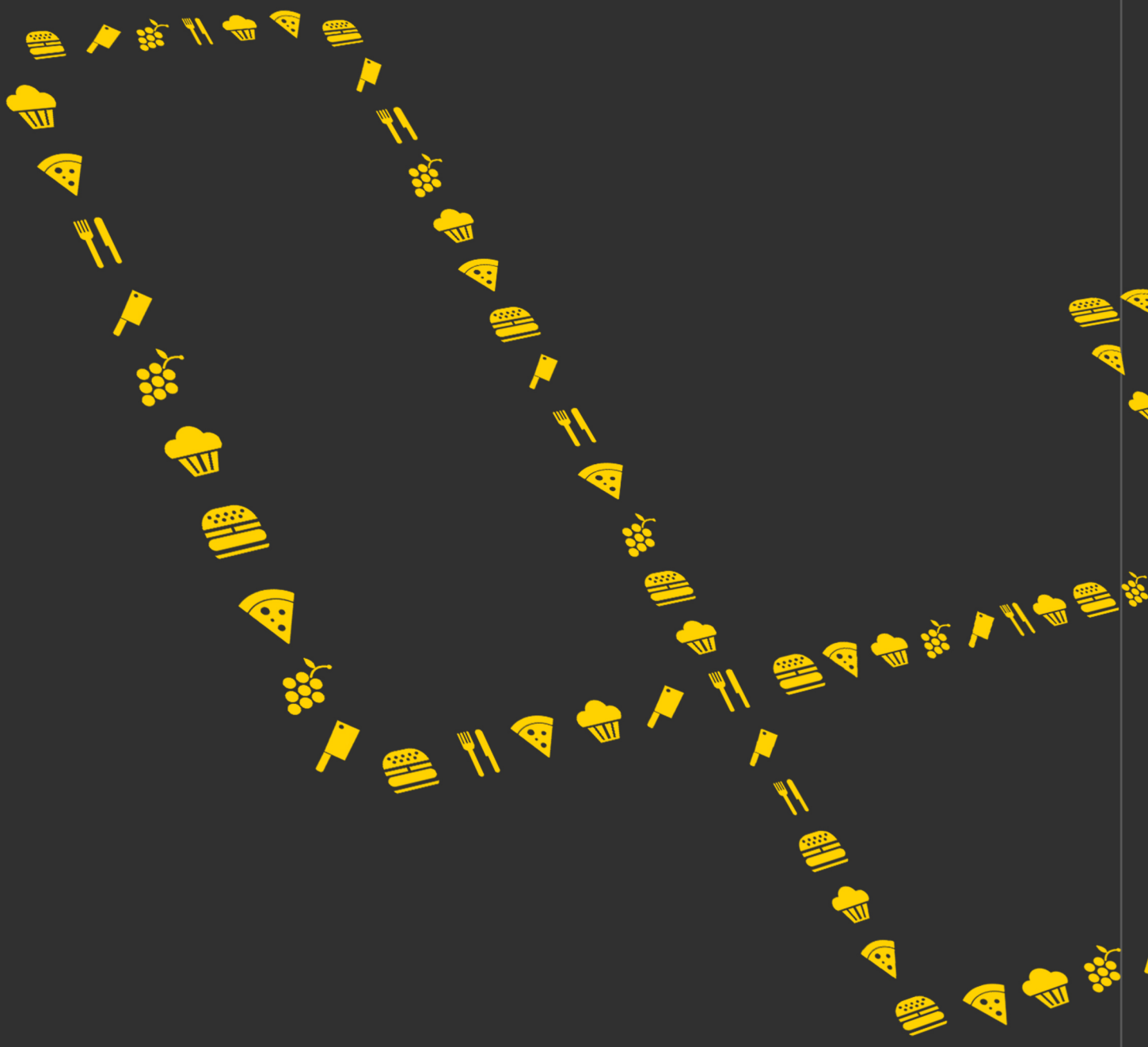

# LARAVEL 4
## COOKBOOK

POR CHRISTOPHER PITT E PEDRO BORGES

# Laravel 4 Cookbook (PT-BR)

## Projetos que você pode desenvolver para aprender sobre o Laravel 4.

Christopher Pitt, Taylor Otwell, ePedro Borges

Esse livro está à venda em http://leanpub.com/laravel4cookbook-pt-br

Essa versão foi publicada em 2014-10-04

# Tweet Sobre Esse Livro!

Por favor ajude Christopher Pitt, Taylor Otwell, ePedro Borges a divulgar esse livro no Twitter!

A hashtag sugerida para esse livro é #laravel4cookbook-pt-br.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#laravel4cookbook-pt-br

# Conteúdo

# Instalando o Laravel 4

O Laravel 4 usa o Composer para gerenciar suas dependências. Você pode instalá-lo seguindo as instruções no site do projeto.

Quando o Composer estiver funcionando, crie uma nova pasta ou navegue até uma já existente e instale o Laravel 4 usando o comando a seguir:

```
composer create-project laravel/laravel ./ --prefer-dist
```

Caso você não tenha instalado o Composer globalmente (embora eu recomendo que você o faça), use este comando:

```
php composer.phar create-project laravel/laravel ./ --prefer-dist
```

Ambos os comandos darão início ao processo de instalação do Laravel 4. Como há muitas dependências para serem resolvidas e baixadas, este processo levará algum tempo para finalizar.

# Autenticação

Se você é como eu, você já deve ter gasto bastante tempo criando algum sistema protegido por senha. Eu tinha tanto pavor disso, que fugia de criar sistemas de autenticação para CMS ou loja virtual. Isso até eu aprender o quanto o Laravel 4 facilita esta tarefa.

> O código deste capítulo está disponível no GitHub.

## Configurando o Banco de Dados

Uma das melhores formas de gerenciar usuários e realizar a autenticação deles é armazenando-os em um banco de dados. O mecanismo padrão de autenticação do Laravel 4 assume que você usará algum tipo banco de dados e oferece dois *drivers*, que são usados para obter e autenticar os usuários em uma aplicação.

### Conectando-se ao banco de dados

Antes de usarmos qualquer um desses *drivers*, precisamos estabelecer uma conexão válida. Configure-a no arquivo `app/config/database.php`. Este é um exemplo de configuração que eu uso para testes:

```php
<?php

return [
  "fetch"       => PDO::FETCH_CLASS,
  "default"     => "mysql",
  "connections" => [
  "mysql" => [
    "driver"    => "mysql",
    "host"      => "localhost",
    "database"  => "tutorial",
    "username"  => "dev",
    "password"  => "dev",
    "charset"   => "utf8",
    "collation" => "utf8_unicode_ci",
    "prefix"    => ""
  ]
  ],
  "migrations" => "migration"
];
```

Este arquivo deve ser salvo como `app/config/database.php`.

Eu omiti comentários, linhas desnecessárias e outras opções supérfluas.

## Driver do banco de dados

O primeiro *driver* oferecido pelo Laravel `4` é chamado de `database`. E como o próprio nome sugere, este é o *driver* que se comunica diretamente com o banco de dados para determinar se as credenciais fornecidas pelo usuário existem e são válidas.

Caso você escolha usar este *driver*, você precisará da seguinte tabela no banco de dados que configuramos anteriormente:

```
CREATE TABLE `user` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `created_at` datetime DEFAULT NULL,
  `updated_at` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

> Neste exemplo e futuramente, eu usarei nomes no singular para as tabelas. Em outra situação eu recomendaria seguir o padrão, mas neste caso, eu quero demonstrar como você pode configurar o nome de uma tabela tanto nas migrações quanto nos modelos.

## Driver Eloquent

O segundo *driver* é chamado de `eloquent`. Eloquent é o nome do ORM presente no *framework* Laravel; sua função é abstrair os dados de um modelo. Ele se parece com o *driver* anterior, pois no fim das contas ele também acessa o banco de dados para verificar se um usuário é autêntico ou não; porém a interface usado por ele é bem diferente.

Se você esta trabalhando em uma aplicação de médio a grande porte usando o Laravel 4, então você tem uma ótima oportunidade para usar o Eloquent para representar objetos do banco de dados. Com isso em mente, eu gastarei algum tempo elaborando como envolver modelos Eloquent no processo de autenticação.

> Se você deseja ignorar tudo sobre o Eloquent, sinta-se à vontade para pular as próximas seções que falam sobre migração e modelos.

## Criando uma migração

Como já estamos usando o Eloquent para gerenciar a comunicação da nossa aplicação com o banco de dados, também usaremos as ferramentas de manipulação de tabelas do Laravel 4.

Para começar, navegue até a raiz do seu projeto e execute o comando a seguir:

```
php artisan migrate:make --table="user" CreateUserTable
```

> A parte `--table="user"` correponde à propriedade `$table = "user"` que definiremos a seguir no modelo `User`.

Este comando gerará o código base para criarmos a tabela `user`; algo assim:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable extends Migration
{
  public function up()
  {
    Schema::table('user', function(Blueprint $table)
    {
      //
    });
  }
  public function down()
  {
    Schema::table('user', function(Blueprint $table)
    {
      //
    });
  }
}
```

> O arquivo será salvo como `app/database/migrations/0000_00_00_000000_CreateUserTable.php`. O seu será um pouco diferente, com os zeros sendo substituídos por outros números.

O nome do arquivo pode até parecer estranho, mas é por um bom motivo. Sistemas de migração são projetados para rodarem em qualquer servidor e a ordem de sua execução é fixa. Tudo isso para permitir que mudanças no banco de dados sejam organizadas em versões.

Agora, vamos criar os campos necessários na tabela dos usuários:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable extends Migration
{
  public function up()
  {
    Schema::create("user", function(Blueprint $table)
    {
      $table->increments("id");

      $table
        ->string("username")
        ->nullable()
        ->default(null);

      $table
        ->string("password")
        ->nullable()
        ->default(null);

      $table
        ->string("email")
        ->nullable()
        ->default(null);

      $table
        ->dateTime("created_at")
        ->nullable()
        ->default(null);

      $table
        ->dateTime("updated_at")
        ->nullable()
        ->default(null);
    });
  }

  public function down()
  {
    Schema::dropIfExists("user");
  }
}
```

> O arquivo deve ser salvo como `app/database/migrations/0000_00_00_000000_CreateUserTable.php`. O seu será um pouco diferente, com os zeros sendo substituídos por outros números.

Aqui nós acrescentamos os campos para o id, nome de usuário, senha e datas de criação e atualização. Há métodos que facilitam a criação de campos para essas datas[1], mas eu prefiro criá-los explicitamente. Todos os campos são *nullable* e o seu valor padrão é *null*.

Nós também acrescentamos o método `drop`, usado sempre que a migração é revertida; o que neste caso seria excluir a tabela `user` caso ela exista.

> O atalho para acrescentar campos para `timestamp` pode ser encontrado na documentação do Laravel.

Esta migração funcionará mesmo que você use apenas o *driver* `database`, mas geralmente ela é usada juntamente com modelos e semeadores[2].

## Criando um modelo

O Laravel 4 já vem com um modelo `User` junto com todos o métodos requeridos por sua interface. Eu modifiquei ele um pouquinho, mas ele é basicamente o mesmo...

[1] *timestamps*, em inglês.
[2] *seeder*, em inglês.

```php
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User
extends Eloquent
implements UserInterface, RemindableInterface
{
  protected $table  = "user";
  protected $hidden = ["password"];

  public function getAuthIdentifier()
  {
    return $this->getKey();
  }

  public function getAuthPassword()
  {
    return $this->password;
  }

  public function getReminderEmail()
  {
    return $this->email;
  }
}
```

> Este arquivo deve ser salvo em `app/models/User.php`.

> Observe que definimos a propriedade `$table = "user"`, seu valor deve ser o mesmo definido em nossa migração.

O modelo `User` extende a classe `Eloquent` e implementa duas interfaces, para assegurar que o modelo é válido para operações de autenticação e recuperação de senha. Vamos aprender mais sobre interfaces no futuro, o importante agora é notar os métodos exigidos por estas interfaces.

O Laravel 4 permite que você use tanto um endereço de e-mail quanto um nome para identificar um usuário, mas o método `getAuthIdentifier()` pode retornar um campo diferente. A interface `UserInterface` especifica um campo para a senha, mas podemos alterá-lo no método `getAuthPassword()`.

O método `getReminderEmail()` retornará o endereço de e-mail do usuário quando ele precisar recuperar sua senha.

No mais, você é livre para personalizar o modelo como quiser sem medo de quebrar o mecanismo de autenticação do *framework*.

## Criando um semeador

O Laravel 4 inclui um semeador de banco de dados. Ele é usado para inserir registros no banco de dados após a primeira migração. Para acrescentar os primeiros usuários ao meu projeto, tenho a seguinte classe semeadora:

```
1  <?php
2
3  class UserSeeder extends DatabaseSeeder
4  {
5    public function run()
6    {
7      $users = [
8        [
9          "username" => "christopher.pitt",
10         "password" => Hash::make("7h3iMOST!53cu23"),
11         "email"    => "chris@example.com"
12       ]
13     ];
14
15     foreach ($users as $user)
16     {
17       User::create($user);
18     }
19   }
20 }
```

> Este arquivo deve ser salvo como `app/database/seeds/UserSeeder.php`.

Meu usuário será criado no banco de dados quando esta classe for executada, mas para que isso aconteça, é preciso acrescentá-la à classe `DatabaseSeeder`:

```php
<?php

class DatabaseSeeder extends Seeder
{
  public function run()
  {
    Eloquent::unguard();
    $this->call("UserSeeder");
  }
}
```

Este arquivo deve ser salvo como `app/database/seeds/DatabaseSeeder.php`.

Agora, quando a classe `DatabaseSeeder` for invocada, ela alimentará a tabela `user` com o meu usuário. Então, se você já configurou sua migração, modelo e já preencheu os detalhes da conexão com o banco de dados, os três comandos a seguir farão com que tudo isso funcione:

```
composer dump-autoload
php artisan migrate
php artisan db:seed
```

O primeiro comando garante que todas as classes novas são carregadas automaticamente. O segundo, cria as tabelas especificadas na migração. Já o terceiro, insere os dados do usuário na tabela indicada.

## Configurando a Autenticação

As opções de configuração do mecanismo de autenticação são esparsas, mas elas permitem alguma personalização.

```php
<?php

return [
  "driver"   => "eloquent",
  "model"    => "User",
  "reminder" => [
    "email"  => "email.request",
    "table"  => "token",
    "expire" => 60
  ]
];
```

Este arquivo deve ser salvo como `app/config/auth.php`.

Todos este itens são importantes e auto-explicativos. O *view* usado para enviar o e-mail de recuperação de senha é definido em `"email" => "email.request"`. O prazo de validade do código secreto[3] é definido em `"expire" => 60` (em minutos).

Preste atenção no *view* especificado em `"email => "email.request"`, estamos pedindo ao Laravel para carregar o arquivo `app/views/email/request.blade.php` ao invés do arquivo padrão, `app/views/email/auth/reminder.blade.php`.

Muita coisa seria melhor se pudéssemos configurá-las sem "tocarmos" nos provedores de serviço; vamos cuidar disso quando for realmente necessário.

# Autenticando

Para permitir que os usuários autênticos entrem em nossa aplicação, vamos criar uma página de entrada onde os usuários poderão digitar suas credenciais. Se elas forem válidas, eles serão redirecionados para a página do seu perfil.

## Criando um view "layout"

Antes de criarmos nossa primeira página, é recomendável abstrair todo o código do *layout* e do estilo. Para este fim, vamos criar um *layout* padrão com vários *includes* usando o processador de *templates* do Laravel, Blade.

Para começar, vamos criar o *layout* padrão:

[3] *token*, em inglês.

```
<!DOCTYPE html>
<html lang=”en”>
  <head>
    <meta charset="UTF-8" />
    <link
      type="text/css"
      rel="stylesheet"
      href="/css/layout.css" />
    <title>
      Tutorial
    </title>
  </head>
  <body>
    @include("header")
    <div class="content">
      <div class="container">
        @yield("content")
      </div>
    </div>
    @include("footer")
  </body>
</html>
```

Este arquivo deve ser salvo como `app/views/layout.blade.php`.

Este *layout* é basicamente um arquivo HTML contendo duas *tags* do Blade. A *tag* `@include()` diz ao Laravel para incluir os arquivos especificados nela. Neste caso, os arquivos `header` e `footer` presentes na pasta `views`.

Você notou que omitimos a extensão `.blade.php`? O Laravel cuida disso automaticamente pra nós. Ele também vinculará os dados de ambos os arquivos “inclusos” neste *view*.

A segunda *tag* do Blade que usamos foi a `@yield()`. Esta *tag* aceita o nome de uma seção e inclui os dados desta seção no resultado final. Os *views* da nossa aplicação extenderão este *layout*. Cada *view* especificará o seu próprio conteúdo na seção `content` para que o seu código seja incorporado ao *layout* padrão. Parece confuso? A seguir você aprenderá como definir uma seção:

```
@section("header")
  <div class="header">
    <div class="container">
      <h1>Tutorial</h1>
    </div>
  </div>
@show
```

Este arquivo deve ser salvo como `app/views/header.blade.php`.

O arquivo `header` contém duas *tags* do Blade. Juntas, elas instruem o Blade a armazenar o seu código na seção apropriada e processá-lo junto com o *template*.

```
@section("footer")
  <div class="footer">
    <div class="container">
      Powered by <a href="http://laravel.com/">Laravel</a>
    </div>
  </div>
@show
```

Este arquivo deve ser salvo como `app/views/footer.blade.php`.

Similarmente, o *include* do rodapé armazena o seu código em uma seção que é processada no *template* imediatamente.

Você deve estar se perguntando: "por que precisamos colocar o código HTML desses arquivos em seções se eles serão processados imediatamente?" É porque fazendo assim, poderemos alterar o seu conteúdo isoladamente. Já vamos ver isso em ação.

```
body
{
  margin      : 0;
  padding     : 0 0 50px 0;
  font-family : "Helvetica", "Arial";
  font-size   : 14px;
  line-height : 18px;
  cursor      : default;
}
a
```

```
11  {
12    color : #ef7c61;
13  }
14  .container
15  {
16    width    : 960px;
17    position : relative;
18    margin   : 0 auto;
19  }
20  .header, .footer
21  {
22    background  : #000;
23    line-height : 50px;
24    height      : 50px;
25    width       : 100%;
26    color       : #fff;
27  }
28  .header h1, .header a
29  {
30    display : inline-block;
31  }
32  .header h1
33  {
34    margin      : 0;
35    font-weight : normal;
36  }
37  .footer
38  {
39    position : absolute;
40    bottom   : 0;
41  }
42  .content
43  {
44    padding : 25px 0;
45  }
46  label, input, .error
47  {
48    clear  : both;
49    float  : left;
50    margin : 5px 0;
51  }
52  .error
53  {
54    color : #ef7c61;
55  }
```

Este arquivo deve ser salvo como `public/css/layout.css`.

Encerramos acrescentando um estilo básico, que é solicitado no elemento `head` do nosso *layout*. Com ele estamos alterando a fonte padrão e a estrutura do *layout*. Sua aplicação não precisa deste arquivo para funcionar, mas não custa nada deixá-la mais atraente, não é mesmo?

## Criando um view para a autenticação

Este *view* é basicamente um formulário onde os usuários poderão digitar suas credenciais.

```
@extends("layout")
@section("content")
  {{ Form::open([
    "route"        => "user/login",
    "autocomplete" => "off"
  ]) }}
    {{ Form::label("username", "Usuário:") }}
    {{ Form::text("username", Input::old("username"), [
      "placeholder" => "john.smith"
    ]) }}
    {{ Form::label("password", "Senha:") }}
    {{ Form::password("password", [
      "placeholder" => "□□□□□□□□□□□"
    ]) }}
    {{ Form::submit("Entrar") }}
  {{ Form::close() }}
@stop
@section("footer")
  @parent
  <script src="//polyfill.io"></script>
@stop
```

Este arquivo deve ser salvo como `app/views/user/login.blade.php`.

A primeira *tag* do Blade, no arquivo `login`, serve para estender o *layout* padrão criado anteriormente. Já a segunda, especifica qual código deve ser incluído na seção `content`. Estas *tags* são a base de todos os *views* que criaremos (exceto o próprio *layout*).

Quando usamos `{{ $e }}`, estamos dizendo ao Laravel para interpretar o seu conteúdo como PHP, é o mesmo que escrever `<?php echo $e; ?>`. Em seguida, nós abrimos o formulário com o método `Form::open()`, passando a rota de destino e outros parâmetros no segundo argumento.

Depois definimos duas etiquetas e três campos. As etiquetas aceitam um nome como primeiro argumento e um texto como segundo argumento. Já o campo de texto aceita um nome como argumento, um valor padrão e parâmetros opcionais como terceiro argumento. O campo de senha aceita um nome como argumento e parâmetros opcionais. E, finalmente, o "botão" *submit* aceita um texto como argumento e outros parâmetros opcionais (ele é parecido com as etiquetas).

Agora só falta fechar o formulário invocando o método `Form::close()`.

> Você pode aprender mais sobre os métodos da classe `Form` na documentação do Laravel.

A última parte do *view* `login` é onde alteramos o rodapé padrão (especificado no `arquivo app/views/footer.blade.php`). Usaremos o mesmo nome de seção, mas desta vez não fecharemos a seção com a *tag* `@show`, usaremos apenas `@stop`, assim como encerramos a seção do conteúdo.

Também usamos a *tag* `@parent` do Blade para dizer ao Laravel que nós queremos que o rodapé padrão também seja exibido. Não estamos substituíndo-o por completo, apenas acrescentamos um *script*.

> Aprenda mais sobre as *tags* do Blade na documentação do Laravel.

O *script* que incluímos se chama polyfill.io. É uma coleção de melhorias para navegadores que, dentre outros recursos, permite o uso do atributo `placeholder` em navegadores onde ele ainda não é suportado.

Agora o nosso *view* `login` está completo, mas ele não possui nenhuma utilidade sem um código no servidor para aceitar os seus dados e retornar algo. Vamos consertar isso!

## Criando uma ação para a autenticação

A ação *login* unirá a lógica de autenticação aos *views*. Você deve estar ansioso para testar o que já fizemos no navegador; mas até agora, nossa aplicação não sabe qual *view* deve ser carregado.

Para começar, precisamos criar uma rota para a ação *login*.

```
1  <?php
2
3  Route::any("/", [
4    "as"   => "user/login",
5    "uses" => "UserController@loginAction"
6  ]);
```

Este arquivo deve ser salvo como `app/routes.php`.

Em uma aplicação nova, o arquivo de rotas retorna apenas um *view* de boas vindas. Vamos alterá-lo para usar um controlador/ação. Não que seja necessário, pois poderíamos executar qualquer lógica diretamente no arquivo de rota, mas desta forma este arquivo não ficaria tão organizado.

Nós especificamos um nome para a rota usando `"as" => "user/login"`; e também o seu destino, `"uses" => "UserController@loginAction"`. Esta rota responderá a qualquer *request* para rota padrão, `/`. O nome `user/login` poderá ser usado na aplicação para se referir a esta rota.

Em seguida, vamos criar o controlador.

```
1  <?php
2
3  class UserController extends Controller
4  {
5    public function loginAction()
6    {
7      return View::make("user/login");
8    }
9  }
```

Este arquivo deve ser salvo como `app/controllers/UserController.php`.

Nós definimos que o `UserController` deve estender a classe `Controller`. Esta classe possui apenas um método chamado `loginAction()`, que foi especificado no arquivo de rotas. Tudo o que este método faz é retornar um *view* chamado *login* para o navegador; isso é suficiente para visualizarmos nosso progresso até aqui.

## Autenticando usuários

Certo, já temos um formulário e agora precisamos ligá-lo ao banco de dados para autenticar os nossos usuários.

```php
<?php

class UserController
extends Controller
{
  public function loginAction()
  {
    if (Input::server("REQUEST_METHOD") == "POST")
    {
      $validator = Validator::make(Input::all(), [
        "username" => "required",
        "password" => "required"
      ]);

      if ($validator->passes())
      {
        echo "Validado com sucesso!";
      }
      else
      {
        echo "Dados inválidos!";
      }
    }

    return View::make("user/login");
  }
}
```

Este arquivo deve ser salvo como `app/controllers/UserController.php`.

Nossa classe `UserController` mudou um pouco. Primeiramente, vamos precisar das informações enviadas ao método `loginAction()`, por isso verificamos a propriedade do servidor `REQUEST_METHOD`. Se este valor for `POST`, podemos assumir que o formulário foi recebido nesta ação e prosseguiremos com o processo de validação.

É comum ver ações separadas para `GET` e `POST` na mesma página. Embora esta forma torne o código mais limpo e não exija a verificação da propriedade `REQUEST_METHOD`, eu prefiro fazer tudo em uma ação só.

O Laravel 4 possui um ótimo sistema de validação e uma das formas de usá-lo é invocando o método `Validator::make()`. O seu primeiro argumento é um *array* contendo os dados a serem

validados e o segundo é um *array* com as regras de validação.

Aqui, nós apenas especificamos que os campos `username` e `password` são obrigatórios[4], porém há muitas outras regras disponíveis (usaremos algumas outras a seguir). A classe `Validator` também possui um método chamado `passes()`, que é usado para verificar se os dados recebidos no primeiro argumento são válidos ou não.

> Às vezes, é melhor armazenar a lógica de validação fora do controlador. Eu costumo colocá-la no modelo, mas você pode até criar uma classe exclusiva para realizar a validação dos dados.

Se você enviar este formulário, ele lhe dirá se os campos obrigatórios foram preenchidos ou não; mas há uma forma mais elegante de exibir este tipo de mensagem…

[4] *required*, em inglês.

```php
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
  public function loginAction()
  {
    $data = [];

    if (Input::server("REQUEST_METHOD") == "POST")
    {
      $validator = Validator::make(Input::all(), [
        "username" => "required",
        "password" => "required"
      ]);

      if ($validator->passes())
      {
        //
      }
      else
      {
        $data["errors"] = new MessageBag([
          "password" => [
            "Usuário e/ou senha inválidos."
          ]
        ]);
      }
    }

    return View::make("user/login", $data);
  }
}
```

Este arquivo deve ser salvo como `app/controllers/UserController.php`.

Com as mudanças acima, estamos usando a classe `MessageBag` para armazenar as mensagens de erro na validação. É mais ou menos assim que a classe de validação armazena seus erros, mas ao invés de exibir uma mensagem de erro para cada campo, estamos exibindo apenas uma mensangem para ambos. Formulários de autenticação são mais seguros assim!

Para exibir esta mensagem de erro, vamos precisar alterar o *view login*.

```
@extends("layout")
@section("content")
  {{ Form::open([
    "route"        => "user/login",
    "autocomplete" => "off"
  ]) }}
    {{ Form::label("username", "Username") }}
    {{ Form::text("username", Input::get("username"), [
      "placeholder" => "john.smith"
    ]) }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password", [
      "placeholder" => "••••••••••"
    ]) }}
    @if ($error = $errors->first("password"))
      <div class="error">
        {{ $error }}
      </div>
    @endif
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
@section("footer")
  @parent
  <script src="//polyfill.io"></script>
@stop
```

Este arquivo deve ser salvo como `app/views/user/login.blade.php`.

Como você pode perceber, nos acrescentamos a verificação da existência de uma mensagem de erro e a exibimos em um elemento `<div>` caso ela exista. Se a validação falhar, você verá esta mensagem abaixo do campo de senha.

## Redirecionando com input

Uma das armadilhas mais comuns nos formulários é que a cada atualização de página o formulário é reenviado. Isso pode ser resolvido com um pouco de mágica do Laravel. Nós vamos armazenar os dados do formulário em uma sessão e redirecioná-los à página de autenticação!

```php
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
  public function loginAction()
  {
    $errors = new MessageBag();

    if ($old = Input::old("errors"))
    {
      $errors = $old;
    }

    $data = [
      "errors" => $errors
    ];

    if (Input::server("REQUEST_METHOD") == "POST")
    {
      $validator = Validator::make(Input::all(), [
        "username" => "required",
        "password" => "required"
      ]);

      if ($validator->passes())
      {
        //
      }
      else
      {
        $data["errors"] = new MessageBag([
          "password" => [
            "Usuário e/ou senha inválidos."
          ]
        ]);

        $data["username"] = Input::get("username");

        return Redirect::route("user/login")
          ->withInput($data);
      }
    }

```

```
47      return View::make("user/login", $data);
48    }
49  }
```

Este arquivo deve ser salvo como `app/controllers/UserController.php`.

A primeira coisa que fizemos foi criar uma nova instância da classe `MessageBag`. Nós fizemos isso porque o nosso *view* ainda verificará se há erros, tendo ou não sido salvos na sessão. Se houver um erro salvo na sessão, substituiremos a instância criada pela armazenada na sessão.

Em seguida ela é acrescentada ao *array* `$data` e enviada ao *view* para ser exibida.

Caso a validação falhe, nós salvamos o nome do usuário no *array* `$data` junto com os erros de validação e redirecionamos tudo de volta para a mesma rota de entrada (usamos o método `withInput()` para armazenar os dados na sessão).

Nosso *view* não foi modificado, mas já podemos atualizar a página sem ser incomodados por aquela mensagem chata do navegador solicitando o reenvio dos dados.

## Autenticando credenciais

O último passo na autenticação é verificar se os dados fornecidos são os mesmos existentes no banco de dados. O Laravel também cuida disso pra nós.

```
1   <?php
2
3   use Illuminate\Support\MessageBag;
4
5   class UserController
6   extends Controller
7   {
8     public function loginAction()
9     {
10      $errors = new MessageBag();
11
12      if ($old = Input::old("errors"))
13      {
14        $errors = $old;
15      }
16
17      $data = [
18        "errors" => $errors
19      ];
20
```

```
21      if (Input::server("REQUEST_METHOD") == "POST")
22      {
23        $validator = Validator::make(Input::all(), [
24          "username" => "required",
25          "password" => "required"
26        ]);
27
28        if ($validator->passes())
29        {
30          $credentials = [
31            "username" => Input::get("username"),
32            "password" => Input::get("password")
33          ];
34
35          if (Auth::attempt($credentials))
36          {
37            return Redirect::route("user/profile");
38          }
39        }
40
41        $data["errors"] = new MessageBag([
42          "password" => [
43            "Usuário e/ou senha inválidos."
44          ]
45        ]);
46
47        $data["username"] = Input::get("username");
48
49        return Redirect::route("user/login")
50          ->withInput($data);
51      }
52
53      return View::make("user/login", $data);
54    }
55  }
```

Este arquivo deve ser salvo como `app/controllers/UserController.php`.

Só precisamos passar os dados enviados através do formulário, `$credencials`, para o método `Auth::attempt()` e, caso suas credenciais sejam válidas, o usuário será autenticado e redirecionado para o seu perfil.

Nós também movemos o código de erros para fora da cláusula `else`. Fazendo assim, esse erro será exibido tanto para erros na validação quanto na autenticação. Uma única mensagem de erro

é algo bom, neste caso.

# Recuperando Senhas

O mecanismo de recuperação de senhas embutido no Laravel 4 é excelente! Vamos configurá-lo para que os nossos usuários possam recuperar suas senhas provendo apenas um endereço de e-mail.

## Criando um view para recuperar senhas

Precisaremos de dois *views* para que os usuários possam recuperar suas senhas. Um para que o usuário digite seu e-mail e receba o código secreto e outro para que ele possa criar uma nova senha.

```
@extends("layout")
@section("content")
  {{ Form::open([
    "route"        => "user/request",
    "autocomplete" => "off"
  ]) }}
    {{ Form::label("email", "E-mail") }}
    {{ Form::text("email", Input::get("email"), [
      "placeholder" => "john@example.com"
    ]) }}
    {{ Form::submit("Recuperar") }}
  {{ Form::close() }}
@stop
@section("footer")
  @parent
  <script src="//polyfill.io"></script>
@stop
```

Este arquivo deve ser salvo como `app/views/user/request.blade.php`.

Este *view* é parecido com o *view* de autenticação, a diferença é que ele possui apenas um campo para o endereço de e-mail.

```
@extends("layout")
@section("content")
  {{ Form::open([
    "url"          => URL::route("user/reset") . $token,
    "autocomplete" => "off"
  ]) }}
    @if ($error = $errors->first("token"))
      <div class="error">
        {{ $error }}
      </div>
    @endif
    {{ Form::label("email", "E-mail") }}
    {{ Form::text("email", Input::get("email"), [
      "placeholder" => "john@example.com"
    ]) }}
    @if ($error = $errors->first("email"))
      <div class="error">
        {{ $error }}
      </div>
    @endif
    {{ Form::label("password", "Password") }}
    {{ Form::password("password", [
      "placeholder" => "••••••••••"
    ]) }}
    @if ($error = $errors->first("password"))
      <div class="error">
        {{ $error }}
      </div>
    @endif
    {{ Form::label("password_confirmation", "Confirm") }}
    {{ Form::password("password_confirmation", [
      "placeholder" => "••••••••••"
    ]) }}
    @if ($error = $errors->first("password_confirmation"))
      <div class="error">
        {{ $error }}
      </div>
    @endif
    {{ Form::submit("Salvar") }}
  {{ Form::close() }}
@stop
@section("footer")
  @parent
  <script src="//polyfill.io"></script>
@stop
```

Este arquivo deve ser salvo como `app/views/user/reset.blade.php`.

Ok, temos um formulário com alguns campos e mensagens de erros. Algo importante para se notar é a mudança na ação deste formulário, mais especificamente o uso de `URL::route()` em combinação com uma variável atribuída ao *view*. Vamos defini-la na ação em instantes, não se preocupe com isso agora.

Eu fiz uma pequena modificação no e-mail que será enviado com o código secreto.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Recuperação de Senha</h1>
    Para recuperar sua senha, preencha este formulário:
    {{ URL::route("user/reset") . "?token=" . $token }}
  </body>
</html>
```

Este arquivo deve ser salvo como `app/views/email/request.blade.php`.

Lembre-se: nós mudamos a configuração para usar este *view* quando o email for enviado ao invés do *view* padrão `app/views/emails/auth/reminder.blade.php`.

## Criando uma ação para recuperar senhas

Para que cada ação se torne acessível, precisamos criar rotas para elas.

```php
<?php

Route::any("/", [
  "as"   => "user/login",
  "uses" => "UserController@loginAction"
]);

Route::any("/request", [
  "as"   => "user/request",
  "uses" => "UserController@requestAction"
]);

Route::any("/reset", [
  "as"   => "user/reset",
  "uses" => "UserController@resetAction"
]);
```

Este arquivo deve ser salvo como `app/routes.php`.

Lembre-se: a rota *request* é para solicitar o código secreto e a rota *reset* é para criar uma nova senha.

Também precisaremos gerar uma nova tabela para armazenar estes códigos; vamos usar o *artisan* para isso.

```
php artisan auth:reminders
```

Este código irá gerar um modelo de migração para a tabela `token`.

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTokenTable
extends Migration
{
  public function up()
  {
    Schema::create("token", function(Blueprint $table)
    {
      $table
        ->string("email")
        ->nullable()
        ->default(null);

      $table
        ->string("token")
        ->nullable()
        ->default(null);

      $table
        ->timestamp("created_at")
        ->nullable()
        ->default(null);
    });
  }

  public function down()
  {
    Schema::dropIfExists("token");
  }
}
```

> Este arquivo deve ser salvo como `app/database/migrations/0000_00_00_000000_CreateTokenTable.php`. O seu será um pouco diferente, com os zeros sendo substituídos por outros números.

Eu modifiquei este modelo um pouquinho, mas na essência ele permaneceu o mesmo. Ele criará uma tabela com os campo `email`, `token` e `created_at`. Esta tabela será usada pelo mecanismo de autenticação do Laravel para gerar e validar os códigos de recuperação de senhas.

Tendo feito isso, agora podemos acrescentar as ações para o nosso recuperador de senhas.

```
public function requestAction()
{
  $data = [
    "requested" => Input::old("requested")
  ];

  if (Input::server("REQUEST_METHOD") == "POST")
  {
    $validator = Validator::make(Input::all(), [
      "email" => "required"
    ]);

    if ($validator->passes())
    {
      $credentials = [
        "email" => Input::get("email")
      ];

      Password::remind($credentials,
        function($message, $user)
        {
          $message->from("chris@example.com");
        }
      );

      $data["requested"] = true;

      return Redirect::route("user/request")
        ->withInput($data);
    }
  }

  return View::make("user/request", $data);
}
```

Extraído de `app/controllers/UserController.php`.

O método `requestAction()` valida os dados enviados pelo formulário praticamente da mesma forma que o método `loginAction()`, mas ao invés de repassá-los para `Auth::attempt()`, eles são repassados para `Password::remind()`. Este método aceita um *array* de credenciais (que normalmente incluem um endereço de e-mail) e opcionamente aceita uma função *callback* que nos permite personalizar o e-mail a ser enviado.

```php
public function resetAction()
{
  $token = "?token=" . Input::get("token");

  $errors = new MessageBag();

  if ($old = Input::old("errors"))
  {
    $errors = $old;
  }

  $data = [
    "token"  => $token,
    "errors" => $errors
  ];

  if (Input::server("REQUEST_METHOD") == "POST")
  {
    $validator = Validator::make(Input::all(), [
      "email"                 => "required|email",
      "password"              => "required|min:6",
      "password_confirmation" => "same:password",
      "token"                 => "exists:token,token"
    ]);

    if ($validator->passes())
    {
      $credentials = [
        "email" => Input::get("email")
      ];

      Password::reset($credentials,
        function($user, $password)
        {
          $user->password = Hash::make($password);
          $user->save();

          Auth::login($user);
          return Redirect::route("user/profile");
        }
      );
    }

    $data["email"] = Input::get("email");

    $data["errors"] = $validator->errors();
```

```
47
48      return Redirect::to(URL::route("user/reset") . $token)
49        ->withInput($data);
50    }
51
52    return View::make("user/reset", $data);
53  }
```

Extraído de `app/controllers/UserController.php`.

O método `resetAction()` é o mesmo. Começamos criando o parâmetro `token` (usado quando a página é redirecionada para preservar o *token* em todos os estágios da página de recuperação). Em seguida pegamos as mensagens de erro antigas, assim como fizemos na página de *login*, e finalmente validamos os dados enviados pelo formulário.

Caso todos os dados passem na validação, vamos passá-los para o método `Password::reset()`. O segundo argumento é a lógica usada para atualizar os registros no banco de dados. Estamos atualizando a senha, salvando o registro e automaticamente autenticando o usuário em nossa aplicação.

Se tudo isso acontecer sem nenhum problema, redirecionaremos o usuário para o seu perfil. Do contrário, vamos redirecioná-lo de volta para a página de recuperação de senha junto com as mensagens de erro.

> Há algo estranho sobre o mecanismo de autenticação; os nomes dos campos `password` e `token` e a validação foram escritos diretamente na função `Password::reset()` e a validação nem mesmo usa a classe `Validation`. Portanto, desde que os seus campos se chamem `password`, `password_confirmation` e `token`, e a sua senha possua mais do que 6 caracteres, você não perceberá nada estranho.
>
> Alternativamente, você pode modificar os nomes dos campos e a validação a ser aplicada no arquivo `vendor/laravel/framework/src/Illuminate/Auth/Reminders/PasswordBroker.php` ou implementar o seu próprio `ReminderServiceProvider` para substituir o do Laravel 4. Os detalhes sobre como criar os seus próprios provedores de serviços vão além do escopo deste livro; você encontrará mais detalhes sobre este assunto no excelente livro do Taylor Otwell: "Laravel: De Aprendiz a Artesão".

> Conforme mencionado anteriormente, você pode definir o prazo de validade do código de confirmação no arquivo `app/config/auth.php`.

Aprenda mais sobre os métodos do mecanismo de autenticação na documentação do Laravel.

Aprenda mais sobre os métodos da classe `Mail` na documentação do Laravel.

# Trabalhando com Usuários Autenticados

Certo, os recursos de autenticação e recuperação de senhas já foram concluídos. Na parte final deste capítulo, usaremos os dados da sessão do usuário em nossa aplicação e protegeremos o acesso às áreas restritas da nossa aplicação.

## Criando uma página de perfil

Para demonstrar os dados do usuário armazenados na sessão a que temos acesso, vamos implementar uma página de perfil.

```
1  @extends("layout")
2  @section("content")
3    <h2>Hello {{ Auth::user()->username }}</h2>
4    <p>Seja bem-vindo ao seu vago perfil.</p>
5  @stop
```

Este arquivo deve ser salvo como `app/views/user/profile.blade.php`.

Este "projeto" de perfil exibe apenas uma informação do usuário; você pode acessar qualquer dado do modelo `User` através do objeto retornado pelo método `Auth::user()`. Qualquer campo ou método definido neste modelo (ou tabela no banco de dados) são acessíveis desta forma.

```
1  public function profileAction()
2  {
3    return View::make("user/profile");
4  }
```

Extraído de `app/controllers/UserController.php`.

O método `profileAction()` é tão simples quanto o seu `view`. Não precisamos passar nenhuma informação do usuário para o `view`, o método `Auth::user()` fará tudo isso por nós!

Mais uma vez, para que esta página se torne acessível, ela precisará de uma rota. Vamos criá-la daqui a pouco, pois agora é um excelente momento para falarmos sobre como restringir o acesso a determinadas páginas da nossa aplicação...

## Criando filtros

O Laravel 4 inclui um arquivo chamado `filters.php`, onde poderemos definir filtros a serem aplicados em um rota (ou grupo delas).

```
1  <?php
2
3  Route::filter("auth", function()
4  {
5    if (Auth::guest())
6    {
7      return Redirect::route("user/login");
8    }
9  });
10
11 Route::filter("guest", function()
12 {
13   if (Auth::check())
14   {
15     return Redirect::route("user/profile");
16   }
17 });
18
19 Route::filter("csrf", function()
20 {
21   if (Session::token() != Input::get("_token"))
22   {
23     throw new Illuminate\Session\TokenMismatchException;
24   }
25 });
```

> Este arquivo deve ser salvo como `app/filters.php`.

O primeiro filtro é para as rotas (ou página se você preferir) que exigem um usuário autenticado para acessá-las. A segunda é exatamente o oposto, para rotas onde o usuário não deve estar autenticado. O último filtro é um que temos usado desde o começo.

Quando usamos o método `Form::open()`, o Laravel inclui automaticamente um campo invisível no formulário. Este campo contém um código de segurança especial que é usado sempre que um formulário é enviado. Não há necessidade de se entender agora porque isto é mais seguro...

> ...mas caso você queira, leia: http://blog.ircmaxell.com/2013/02/preventing-csrf-attacks.html.

Para que estes filtros tenham efeito, vamos modificar nosso arquivo de rotas.

```
1  <?php
2
3  Route::group(["before" => "guest"], function()
4  {
5    Route::any("/", [
6      "as"   => "user/login",
7      "uses" => "UserController@loginAction"
8    ]);
9
10   Route::any("/request", [
11     "as"   => "user/request",
12     "uses" => "UserController@requestAction"
13   ]);
14
15   Route::any("/reset", [
16     "as"   => "user/reset",
17     "uses" => "UserController@resetAction"
18   ]);
19 });
20
21 Route::group(["before" => "auth"], function()
22 {
23   Route::any("/profile", [
24     "as"   => "user/profile",
25     "uses" => "UserController@profileAction"
26   ]);
27
28   Route::any("/logout", [
```

```
29      "as"   => "user/logout",
30      "uses" => "UserController@logoutAction"
31    ]);
32  });
```

Este arquivo deve ser salvo como `app/routes.php`.

Para proteger certas partes da nossa aplicação, nós agrupamos as rotas usando o método `Route::group()`. O primeiro argumento nos permite especificar quais filtros devem ser aplicados às rotas contidas neste grupo. Nós também agrupamos as páginas que o usuário não precisa acessar quando já estiver autenticado. Fizemos exatamente o oposto para a página do perfil, pois somente aos usuários autenticados é permitido visualizar seus perfis.

## Criando uma ação de saída

Para testarmos essas novas medidas de segurança (e assim encerrarmos esta seção), precisamos criar um método chamado `logoutAction()` e acrescentar um link ao cabeçalho para que os usuários possam sair da aplicação.

```
1  public function logoutAction()
2  {
3    Auth::logout();
4    return Redirect::route("user/login");
5  }
```

Extraído de `app/controllers/UserController.php`.

O método `logoutAction()` invoca outro método, `Auth::logout()` para encerrar a sessão do usuário e redirecioná-lo à tela de *login*. Mole mole!

O novo cabeçalho[5] ficará assim:

[5] *header*, em inglês.

```
@section("header")
  <div class="header">
    <div class="container">
      <h1>Tutorial</h1>
      @if (Auth::check())
        <a href="{{ URL::route("user/logout") }}">
          sair
        </a>
        |
        <a href="{{ URL::route("user/profile") }}">
          perfil
        </a>
      @else
        <a href="{{ URL::route("user/login") }}">
          entrar
        </a>
      @endif
    </div>
  </div>
@show
```

Este arquivo deve ser salvo como `app/views/header.blade.php`.