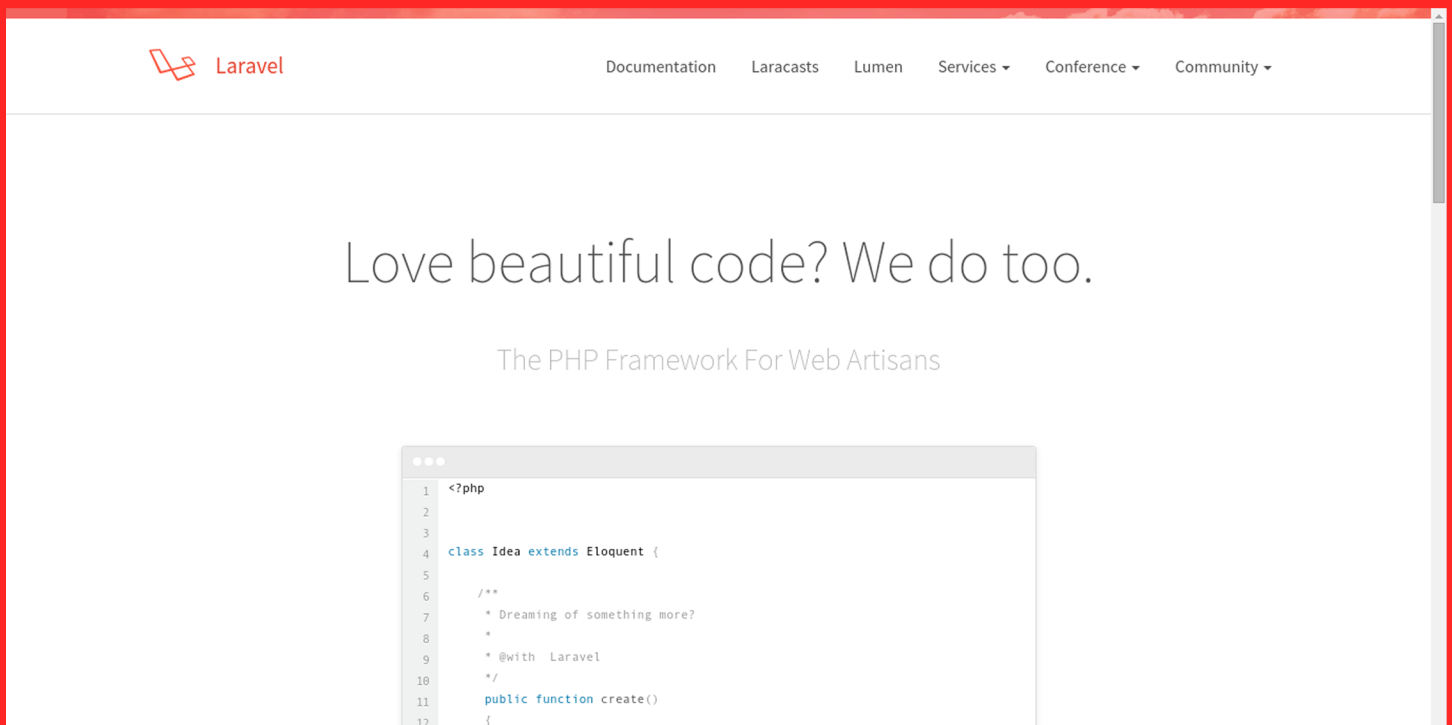
laravel essencial

apostila

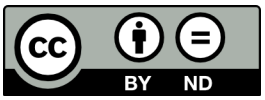
versão 5.1



Laravel 5.1 Essencial

O framework PHP para artesãos Web

Erik Figueiredo



Dedico este livro a minha esposa e filha pela paciência e apoio.

Conteúdo

Introdução

Ambiente de desenvolvimento

Atualmente estou usando o Ubuntu com PHP 5.6.8, mas sempre mantenho atualizado na versão mais rescente e o banco de dados MySql, claro que você deve usar o que já está acostumado, por exemplo, se usa Windows com Xampp ou Wamp, se usa Mac ou qualquer outro tipo de configuração de ambiente de desenvolvimento, não importa, só peço que verifiquem se o seu ambiente está de acordo com os seguintes pré-requisitos, que são obrigatórios para o Laravel rodar sem problemas.

- PHP >= 5.5.9 (lembre-se que recomendei a versão >= 5.6)
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

Outro ponto importante é que vamos precisar do Composer[1].

Para instalar o composer vá até o site oficial (https://getcomposer.org/[2]) e escolha como prefere fazer o Download, não importa como será feito, desde que você tenha o arquivo.

Existem duas formas de executar o Composer, a primeira é conhecida como instalação local, nela o arquivo `composer.phar` fica no diretório que você baixou ou que vai começar o seu projeto, todos os comando são executados usando o PHP (via terminal) seguido do arquivo `phar`, assim.

```
1 php composer.phar [comando]
```

A segunda forma é chamada de instalação global, ou seja, você tem o Composer configurado no sistema operacional de forma a conseguir usar os comandos em qualquer diretório sem que precise se preocupar em baixar toda vez que vai usar, a utilização é simplificada também.

```
1 composer [comando]
```

Além das diferenças apontadas acima (local e forma de usar) não existem outras vantagens e desvantagens em relação a utilização de uma forma ou de outra, eu prefiro a global, já que uso muito o Composer no meu dia a dia.

Existe um passo a passo para instalar o Composer de forma global no Windows, Linux, Unix e OSX na documentação oficial.

[1] https://getcomposer.org/
[2] https://getcomposer.org/

- Instalar o Composer globalmente no Windows: https://getcomposer.org/doc/00-intro.md#installation-windows[3]
- Instalar o Composer Globalmente no Linux, Unix e OSX: https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx[4]

Existem até instaladores para estes sistemas operacionais nos links acima.

Não se preocupe, falaremos mais sobre o composer no próximo capítulo.

O que é Orientação a Objetos?

Note que só com esta sessão já teriamos conteúdo suficiente para escrever dois livros, então vou tentar simplificar o assunto e focar apenas no necessário para usar o Laravel, claro que posso me empolgar e ir além, não estranhe se isso acontecer.

> A orientação a objetos é um modelo de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos. – *Wikipedia*

Orientação a objetos define que cada parte da aplicação pode ser definida como um objeto que possui atributos e métodos, uma definição mais simples para um objeto seria dizer que ele possui atributos e ações, sendo que cada ação ou método é definido usando a palavra-chave `function` enquanto um atributo é uma variável do objeto ou classe. O código a seguir mostra um claro exemplo do que é um objeto.

```
1  <?php
2
3  class Objeto {
4      $atributo;
5
6      function metodoAcao() {
7          return 'Hello World';
8      }
9  }
```

Este bloco de código, por sí só, não traz nenhum resultado direto a aplicação, a classe ou objeto em questão ficará disponível para ser usado durante a execução da aplicação, veja o exemplo anterior com a utilização.

[3] https://getcomposer.org/doc/00-intro.md#installation-windows
[4] https://getcomposer.org/doc/00-intro.md#installation-linux-unix-osx

```php
<?php

class Objeto {
    $atributo;

    function metodoAcao() {
        return 'Hello World';
    }
}

$objeto = new Objeto;
echo $objeto->metodoAcao();
```

Na penúltima linha do código acima usamos a palavra-chave `new` para carregar o objeto dentro da variável `$objeto` e então ter acesso aos seus recursos na linha seguinte, este processo é chamado de instanciação, é comum dizer que 'vamos instanciar a classe tal'.

Embora funcional este objeto ainda não está escrito da melhor forma possível, um objeto escrito segundo os padrões modernos do PHP deve obrigatóriamente seguir a PSR-2 (o que nos força a usar a PSR-4 e a PSR-1).

Este é um típico objeto escrito no PHP de acordo com as práticas modernas.

```php
<?php

namespace WebDevBr\Orientacao\Objetos;

class Objeto
{
    public $atributo;

    public function metodoAcao()
    {
        return 'Hello World';
    }
}
```

E a correta utilização.

```php
<?php

/* Carregamento do arquivo da classe (com include/require, comando 'use' ou auto\
load) */

$objeto = new WebDevBr\Orientacao\Objetos\Objeto;
echo $objeto->MetodoAcao();
```

Cada objeto deve estar em um arquivo único, ter um namespace e ser carregado por um autoloader de acordo com as PSRs, se não está familiarizado com estes conceitos recomendo que invista algum tempo no site oficial do grupo *PHP FIG*, mantedor das PSRs, embora não seja obrigatório para o entendimento deste livro, com certeza vai ajudar muito.

Aqui o site do PHP FIG http://www.php-fig.org/[5].

Vantagens da orientação a objetos!

A principal vantagem de se trabalhar com orientação a objetos é a reusabilidade do código, focamos em escrever menos e reaproveitar mais o que já foi escrito, essa prática leva a códigos mais limpos, menores e fáceis de manter, além da segurança já que cada classe só tem acesso aos métodos/atributos dela ou que permitirmos.

A estrutura da orientação a objetos

A orientação a objetos está estabelecida sobre 4 pilares básicos de desenvolvimento que serão listados a seguir.

Herança

Herança, na prática, é o que faz com que uma classe tenha acesso a recursos de outra, ta, vai um pouco além, mas já é um bom começo, vamos moldar isso.

Imagine que você tem uma classe que retorna a letra *A* (não é criativo, mas é funcional, e isso é tudo o que precisamos agora).

[5] http://www.php-fig.org/

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

class A
{
        public function getLetterA()
        {
                return 'A';
        }
}
```

E também temos uma classe B que vai herdar a classe A.

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

class B extends A
{
        public function getLetterB()
        {
                return 'B';
        }
}
```

E por fim uma classe C que herda a classe B.

<?php

```php
namespace WebDevBr/Orientacao/Objetos;

class C extends B
{
        public function getLetterC()
        {
                return 'C';
        }
}
```

Já deu pra entender que sempre que formos herdar uma classe usamos o `extends`, mas o que realmente aconteceu ali? Sempre que herdarmos uma classe teremos acesso a todos os métodos e atributos que nos forem permitidos, veremos mais sobre permitir e proibir acesso a métodos e atributos quando falarmos de **Encapsulamento** logo mais a frente. Veja agora como usamos classes com herança.

```php
<?php

/* Carregamento dos arquivos das classe (com include/require ou autoload) */

$c = new WebDevBr\Orientacao\Objetos\C;
echo $c->getLetterA().'<br>';
echo $c->getLetterB().'<br>';
echo $c->getLetterC();
```

Além disso ainda podemos executar os métodos internamente na classe usando a variável `$this`.

<?php

```
namespace WebDevBr/Orientacao/Objetos;

class C extends B
{
        public function getLetterB()
        {
                return 'C';
        }

        public function getLetters()
        {
                $str = $this->getLetterA().'<br>';
                $str .= $this->getLetterB().'<br>';
                $str .= $this->getLetterC();
        }
}

$c = new WebDevBr\Orientacao\Objetos\C;
echo $c->getLetters();
```

Isto é o que chamamos de herança vertical, quando uma classe extende outra, que extende outra, e outra... Ainda temos a herança horizontal, representada pelos `traits`, que não falaremos aqui já que não será usado, e a herança múltipla que não é aplicado no PHP, por enquanto.

Abstração

Muitas vezes precisamos criar métodos em várias classes que são comuns a vários outros objetos, muitas vezes o código é exatamente o mesmo em todos os lugares que é usado, nestes casos temos recursos para nos auxiliar da forma correta, que tal criarmos uma classe intermediária que guarde este(s) método(s).

Vamos imaginar um novo cenário, em que as letras não vão herdar uma a outra (faz até mais sentido), e todas vão retornar a sua própria letra, mas pense, e se em algum momento tivermos que alterar, por exemplo, para letras minúsculas, são 26 classes, uma para cada letra.

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

abstract class Letters
{
        public function getLetter()
        {
                return $this->letter;
        }
}
```

Este novo objeto não pode ser instanciado, é uma classe abstrata, só serve para herdar.

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

class A extends Letters
{
        public $letter = 'A';
}

$objeto = new WebDevBr\Orientacao\Objetos\A;
echo $objeto->getLetter();
```

Agora estamos organizando as coisas, mas só arranhamos a superfície, abstração vai além disso, você também pode forçar a classe que filha (que está herdando a `Letter`, que é a classe pai) a criar métodos, por exemplo:

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

abstract class Letters
{
        public function getLetter()
        {
                return $this->letter;
        }

        public function setLetter($letter);
}
```

Quando eu for criar uma classe ela deve, obrigatóriamente, implementar um método `setLetter($letter)`.

```php
<?php

namespace WebDevBr/Orientacao/Objetos;

class Alphabet extends Letters
{
        protected $letter;

        public function setLetter($letter)
        {
                if (!preg_match('^[a-Z]{1}$', $letter))
                        thown new Exception('Isso não é uma letra');
                $this->letter = $letter;
        }
}

$objeto = new WebDevBr\Orientacao\Objetos\Alphabet;
$objeto->setLetter('A');
echo $objeto->getLetter();
```

Essa prática é um padrão de projeto chamado de Template Method, que informa a estrutura de uma classe enquanto implementa métodos, quando não implementamos métodos (uma `interface`) chamamos `Strategy`, claro, é uma explicação superficial.

Encapsulamento

Na verdade eu gosto da definição da Wikipedia, mais claro que isso...

> Encapsulamento vem de encapsular, que em programação orientada a objetos significa juntar o programa em partes, o mais isoladas possível. A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações.

Além disso também podemos citar a visibilidade dos métodos e atributos, que são 3.

- `public`: Pode ser acessado de qualquer lugar, é o valor padrão
- `private`: Só pode ser acessado dentro do objeto dono do atributo/método
- `protected`: Como o `private`, mas também pode ser acessado a partir de objetos filhos (que herdaram a classe)

Polimorfismo

Segundo o Google a palavra polimorfismo quer dizer.

> Qualidade ou estado de ser capaz de assumir diferentes formas.

Segundo a Wikipedia.

> Na programação orientada a objetos, o polimorfismo permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato). O termo polimorfismo é originário do grego e significa "muitas formas" (poli = muitas, morphos = formas). O polimorfismo é caracterizado quando duas ou mais classes distintas tem métodos de mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto. Uma das formas de implementar o polimorfismo é através de uma classe abstrata, cujos métodos são declarados mas não são definidos, e através de classes que herdam os métodos desta classe abstrata.

Para mais informações indico a leitura direta na Wikipedia, https://pt.wikipedia.org/wiki/Polimorfismo[6].

[6] https://pt.wikipedia.org/wiki/Polimorfismo

Continuar aprendendo

Aqui é só arranhei a superficie, existem muito mais além, tente pesquisar mais sobre o assunto ou ver o curso de PHP Orientado do WebDevBr, vai te dar um panorama mais prático.

Além disso é interessante que você pesquise sobre SOLID e Object Calisthenics, o primeiro são práticas que vão organizar sua aplicação de forma a poder crescer sem dificuldades, enquanto que o segundo são exercícios que e ajudaram a escrever códigos mais bonitos, claro que são definições genéricas.

O que é MVC

Práticas modernas do PHP exigem estudo e preparação, e o padrão de projeto que merece muita atenção é o MVC. Muita gente conhece este padrão através dos frameworks (isso não é um problema, eu mesmo estou neste grupo), mas ir a fundo é essencial para evitar erros e falar coisas como:

> Seu MVC está errado, o controller está maior que o model.

Este é um erro de definição já que em nenhum lugar está escrito que a quantidade de linhas define o padrão MVC, mas vamos entender isto melhor?

Model

Model é onde fica a lógica da aplicação. Só isso.

Vai disparar um e-mail? Validar um formulário? Enviar ou receber dados do banco? Não importa. A model deve saber como executar as tarefas mais diversa, mas não precisa saber quando deve ser feito, nem como mostrar estes dados.

View

View exibe os dados. Só isso.

View não é só o HTML, mas qualquer tipo de retorno de dados, como PDF, Json, XML, o retorno dos dados do servidor RESTFull, os tokens de autenticação OAuth2, entre outro. Qualquer retorno de dados para uma interface qualquer (o navegador, por exemplo) é responsabilidade da view. A view deve saber renderizar os dados corretamente, mas não precisa saber como obtê-los ou quando renderizá-los.

Controller

O controller diz quando as coisas devem acontecer. Só isso.

É usado para intermediar a model e a view de uma camada. Por exemplo, para pegar dados da model (guardados em um banco) e exibir na view (em uma página HTML), ou pegar os dados de um formulário (view) e enviar para alguém (model). Também é responsabilidade do controller cuidar das requisições (request e response) e isso também inclui os famosos middlewares (Laravel, Slim Framework, Express, Ruby on Rails, etc.). O controller não precisa saber como obter os dados nem como exibi-los, só quando fazer isso.

O que é um framework

> Um Framework ou arcabouço conceitual é um conjunto de conceitos usado para resolver um problema de um domínio específico. Framework conceitual não se trata de um software executável, mas sim de um modelo de dados para um domínio. – *Wikipedia*

Em outras palavras um framework é uma estrutura base para se começar um projeto, a grande diferença entre framework e biblioteca é que o framework vai ditar a organização/fluxo da aplicação. É comum ver a comparação no Javascript, mais especificamente no famoso embate AngularJs x Jquery, sendo que o AngularJs já te ajuda com a estrutura do seu projeto, enquanto o Jquery (que é usado internamente pelo AngularJs) apenas te da as ferramentas para serem usadas, você tem que se organizar.

Framework Full-Stack VS Micro frameworks

A maioria dos frameworks atuais são divididos em duas subclasses, os micro e os full-stack.

Micro frameworks são focados em resolver um único problema, como é o caso do Silex, Slim Framework e Lumen (este último baseado no Laravel), e muitas vezes só te disponibilizam uma camada de controller e view (muita vezes esta view é simples, sem muitos recursos), se você quer uma camada de model, emails, logs ou outros recursos terá que instalar a parte.

Frameworks full-stack são os caras mais completos e incrementados, eles tentam te trazer todas as ferramentas e resolver todos os problemas. É o caso do Laravel, CakePHP, Zend Framework, Symfony e tantos outros por ai.

É dificil definir quem é melhor ou pior, micro frameworks trazem uma visão simplista, você tem o mínimo possível e se mantém usando apenas o necessário, não preciso falar das vantagens dos frameworks full-stack, afinal, se você está lendo este livro é porque já tomou sua decisão.

O que vamos desenvolver?

Durante o curso vamos criar uma loja virtual, vou chama-la de L-Commerce (o nome está aberto a sugestões, rsrs). Vamos criar uma administração, listagem de produtos, organizar por categorias, um carrinho de compras e até uma integração com PagSeguro e Correios.

Para isso vamos precisar criar uma área de administração protegida com senha, cadastro de usuário, produtos e categorias, relacionamento e no final teremos também toda a parte de integração com os serviços externos (PagSeguro e Correios), muita coisa pra mexer em!

Vamos por a mão na massa?

Aonde conseguir ajuda?

Estes são alguns locais que você vai querer visitar.

- http://laravel.com/docs/5.1[7] - Documentação oficial (procure sempre aqui primeiro)
- http://laravel-docs.artesaos.org/[8] - Documentação traduzida
- https://www.facebook.com/groups/laravelbrasil/[9] - Slack da comunidade Laravel Brasil
- http://slack.laravel.com.br/[10] - Comunidade no Facebook

Você também pode entrar em contato comigo através de um dos canais de comunicação divulgados no capítulo anterior.

Projeto final

Todo o código do projeto já finalizado pode ser baixado em https://github.com/erikfig/Curso-de-Laravel-5.1[11]. ___

[7] http://laravel.com/docs/5.1
[8] http://laravel-docs.artesaos.org/
[9] https://www.facebook.com/groups/laravelbrasil/
[10] http://slack.laravel.com.br/
[11] https://github.com/erikfig/Curso-de-Laravel-5.1

Preparando o Laravel

Instalando o Laravel 5

Antes de instalar o Laravel 5 tenha certeza de estar tudo de acordo com o capítulo 1 > Ambiente de desenvolvimento.

Para começar vamos criar um projeto do Laravel usando o Composer

```
1  composer create-project --prefer-dist laravel/laravel path
```

O path ali no final é o diretório que será instalado o Laravel 5, caso já esteja no diretório destino tente usar `./` (no Linux):

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

Caso não informe o path um diretório chamado **laravel** será criado.

O comando `create-project` baixa um componente (o esqueleto do Laravel neste caso) como seria com o `git clone` ou o simples baixar e descompactar e e seguida faz um `composer install`, trazendo assim as dependências do esqueleto (o Laravel, neste caso).

Rodando o PHP Built-In Server com Artisan

Para usar o PHP Built-in Server do PHP é muito simples, na raiz do projeto execute:

```
1  php artisan serve
```

O Laravel vai, na verdade, executar o arquivo em *vendor/laravel/framework/src/Illuminate/Foundation/Console/ServeCommando.php, e disparar o método `fire()`, que roda algo como:

```
1  php -S localhost:8000 -t public /server.php
```

Ou seja, é um atalho para o PHP Built-In Server.

Ao abrir o navegador com o endereço *http://localhost:8000* (informado no terminal após rodar o `php artisan serve`) você deve se deparar com a tela de boas vindas do Laravel 5.

Permissões de diretórios

Pode ser que após isso você ainda receba mensagens de erro relacionadas a permissão, acontece que o Laravel 5 precisa de permissão de leitura e escrita nos diretórios:

- storage
- bootstrap/cache

Na verdade é bem simples, na maioria dos sistemas operacionais basta rodar os comandos abaixo (respeitando o separador de diretórios, por exemplo, no Windows é barra invertida [\]).

```
1 chmod 0777 -R storage
2 chmod 0777 -R bootstrap/cache
```

Existem outras formas de se fazer isso, o importante é que ao final do processo você tenha permissão de leitura e escrita em ambos os diretórios recursivamente.

Configurações iniciais

O Laravel 5 usa arquivos com a extensão *.env* para setar as configurações internas da aplicação como banco de dados, cache, email e a chave de segurança (entre outras), isso é possível graças a biblioteca DotEnv[12] criado por *Vance Lucas*.

Como usamos o Composer, o arquivo *.env* foi criado automaticamente com base no *.env.example*, ambos na raiz do projeto, e agora podemos apenas editar os dados dentro dele, note que este arquivo não deve ser enviado para o servidor de produção, você vai criar um novo lá, então se você usa Git não esqueça de adicionar ao seu *.gitignore* ou excluir da fila do FTP.

A chave da aplicação já deve ter sido gerado pelo Composer, podemos consultar no item *APP_KEY* do seu arquivo *.env*, se não encontrar uma string randômica formada por 32 caracteres entre letras e números então você deve rodar o comando a seguir.

```
1 php artisan key:generate
```

Sem esse código único e randômica sua aplicação não está segura.

Com exceção do *.env*, todos os arquivos de configuração do Laravel 5 estão dentro do diretório *config*, e você pode facilmente editar qualquer um que precise lá dentro, abra o arquivo *config/app.php*, por exemplo, lá existe as configurações de debug (`true` se a aplicação está em desenvolvimento e `false` se está em produção), location (idioma), timezone (horário) e várias outras, vamos ver como preceder caso queira configurar algo, por exemplo, o banco de dados.

[12] https://github.com/vlucas/phpdotenv

Configurando o banco de dados

Ainda dentro do arquivo *.env* vamos encontrar 4 linhas, parecidas com estas:

```
DB_HOST=dev.local
DB_DATABASE=curso_laravel
DB_USERNAME=root
DB_PASSWORD=123
```

Acredito que dispensa maiores detalhes, mas vou explicar mesmo assim.

É ai que você vai inserir as informações para acesso ao banco de dados, provavelmente você já tem estes dados e é capaz de criar um banco de dados vazio por conta própria, de qualquer forma se precisar de ajuda nesta etapa pode solicite ajuda.

Aqui uma descrição do que cada quer dier:

- DB_HOST: O servidor de banco de dados
- DB_DATABASE: O nome do seu banco
- DB_USERNAME: O usuário de acesso
- DB_PASSWORD: A senha (pode ficar em branco se você não setou uma senha)

Após informar os dados no arquivo .env não esqueça de apache o cache de configurações do banco com o comando abaixo.

```
php artisan config:clear
```

Você também pode informar estes dados no arquivo *config/database.php*, a parte que vamos alterar deve se parecer com:

```
/* .. código anterior */
'connections' => [

    'sqlite' => [
        'driver'   => 'sqlite',
        'database' => storage_path('database.sqlite'),
        'prefix'   => '',
    ],

    'mysql' => [
        'driver'    => 'mysql',
```

```
12          'host'      => env('DB_HOST', 'localhost'),
13          'database'  => env('DB_DATABASE', 'forge'),
14          'username'  => env('DB_USERNAME', 'forge'),
15          'password'  => env('DB_PASSWORD', ''),
16          'charset'   => 'utf8',
17          'collation' => 'utf8_unicode_ci',
18          'prefix'    => '',
19          'strict'    => false,
20      ],
21  /* .. código posterior */
```

Note que em vez de informar o servidor (DB_HOST), banco de dados (DB_DATABASE), usuário (DB_USERNAME) e senha (DB_PASSWORD) o arquivo chama o método *env()* com o nome da configuração usada no arquivo *.env* como primeiro parâmetro e um valor padrão como segundo parâmetro. Ele está resgatando os dados do arquivo .env e caso não encontre o segundo parâmetro é o que será usado.

Caso tenha alguma dificuldade em usar o .env ainda podemos substituir o método pelo valor final, firaria assim:

```
1   /* .. código anterior */
2   'connections' => [
3
4       'sqlite' => [
5           'driver'   => 'sqlite',
6           'database' => storage_path('database.sqlite'),
7           'prefix'   => '',
8       ],
9
10      'mysql' => [
11          'driver'    => 'mysql',
12          'host'      => 'dev.local',
13          'database'  => 'curso_laravel',
14          'username'  => 'root',
15          'password'  => '123',
16          'charset'   => 'utf8',
17          'collation' => 'utf8_unicode_ci',
18          'prefix'    => '',
19          'strict'    => false,
20      ],
21  /* .. código posterior */
```

Logo vamos descobrir se as configurações estão corretas.

Conhecendo o banco de dados do nosso projeto

O nosso projeto tera as seguintes tabelas:

- ***categories*** **- Listagem de categorias**
  - id - indíce único
  - title - título da categoria
  - created_at - data de criação
  - updated_at - data de atualização
- ***products*** **- Listagem de produtos**
  - id - indíce único
  - title - título do produto
  - body - descrição do produto
  - value - valor do produto
  - qtd - quantidade disponível do produto
  - url - url amigável do produto
  - created_at - data de criação
  - updated_at - data de atualização
- ***category_product*** **- Relaciona uma categoria com um produto**
  - id - indíce único
  - product_id - O produto
  - category_id - A categoria
- ***users*** **- Gerencia os usuário (já vai ser criado pelo Laravel)**
  - id - indíce único
  - name - nome do usuário
  - email - email do usuário
  - password - senha do usuário
  - created_at - data de criação
  - updated_at - data de atualização
- ***password_resets*** **- Gerencia os resets de senha (já vai ser criado pelo Laravel)**
  - email - email da conta que terá a senha resetada
  - token - código para confirmar a autenticidade da solicitação
  - created_at - data de criação
  - updated_at - data de atualização

Notou que existem alguns que se repetem, vamos entende-los

O campo id

O campo id é a identificação única do registro em questão, vamos usar muito ele na administração do site para abrir um registro ou para relacionar os produtos com as categorias, ele será preenchido automaticamente pelo banco de dados com um número que não vai se repetir.

O campo created_at

O campo created_at é preenchido automaticamente pelo Laravel armazenando a data e hora da criação do registro em questão.

O campo updated_at

O campo updated_at é preenchido automaticamente pelo Laravel armazenando a data e hora que o registro foi atualizado.

Criando arquivos de Migration

Uma das primeiras coisas que faço quando começo a trabalhar em um novo projeto, independente do framework ou linguagem que vou usar, é modelar o banco de dados (acho que todo mundo) e de alguma forma passar a responsabilidade de criar e manter todas as tabelas e registros ao PHP. Isso facilita o deploy e eu não tenho que ficar abrindo o banco de dados para fazer alterações, esse é o processo de migração ou `migration`.

O migration cria uma linha do tempo com os detalhes de criação e posteriores alterações e evoluções do seu banco de dados, assim você pode voltar atraz ou até recriar todo o banco com poucos comandos, é muito prático.

O Laravel 5 já vem com duas tabelas prontas para serem criadas, a de usuários e a de reset de senhas, os arquivos ficam dentro de */database/migrations*. Para criar as tabelas que o Laravel traz basta abrir o diretório raiz do projeto no terminal e rodar o comando a seguir:

```
1  php artisan migrate
```

Claro, você já tem que ter uma tabela criada e o acesso ao banco de dados configurado (no capítulo anterior) ou vai receber um erro.

Até então sem muitas novidades, mas como criar uma nova tabela? Simples, primeiro precisamos criar um novo arquivo de migration, vamos criar uma tabela para nossos produtos, vou chamar de products.

php artisan make:migration products

O *products* foi o nome que eu dei ao arquivo, ele já deve estar junto com os que já existiam.

Dentro do arquivo você vai ter dois métodos, um chamado up() e outro chamado down(), o primeiro executa alterações, o segundo desfaz, veja como é simples:

- up() - cria a tabela
- down() - remove a tabela

Outro exemplo seria o comando up() criar um campo, o down() remove o campo, é simples.

Se você olhar o arquivo que cria a tabela *users* vai matar de primeira como deve fazer com a **products**, mas vou ajudar. Isto vai dentro do método up();

```
Schema::create('products', function (Blueprint $table) {
    $table->increments('id');
    $table->string('title');
    $table->longText('body');
    $table->decimal('value', 11, 2);
    $table->integer('qtd');
    $table->string('url')->unique();
    $table->timestamps();
});
```

E este dentro do down();

```
Schema::drop('products');
```

Na hora de criar os campos usei alguns métodos específicos para cada tipo, por exemplo, para um `varchar` usei `string()`, para um `text` usei `longText()`, para ver a lista completa de uma olhada aqui: http://laravel.com/docs/5.1/migrations#writing-migrations[13].

O `increments()` cria uma chave primária e o `timestamps()` cria as tabelas *created_at* e *updated_at*.

Foi fácil né, rode o `php artisan migrate` novamente pra ver sua nova tabela no banco de dados.

Aqui os outros dois arquivos de migration para as tabelas *categories* e *category_product*, useo o comando `php artisan make:migration [nome_do_arquivo_com_caracteres_minusculos_e_underlines]`.

```
<?php

//categories

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

class Categories extends Migration
{
    /**
     * Run the migrations.
```

[13] http://laravel.com/docs/5.1/migrations#writing-migrations

```
12          *
13          * @return void
14          */
15         public function up()
16         {
17             Schema::create('categories', function (Blueprint $table) {
18                 $table->increments('id');
19                 $table->string('title');
20                 $table->timestamps();
21             });
22         }
23
24         /**
25          * Reverse the migrations.
26          *
27          * @return void
28          */
29         public function down()
30         {
31             Schema::drop('categories');
32         }
33     }
34
35     <?php
36
37     //category_product
38
39     use Illuminate\Database\Schema\Blueprint;
40     use Illuminate\Database\Migrations\Migration;
41
42     class CategoryProduct extends Migration
43     {
44         /**
45          * Run the migrations.
46          *
47          * @return void
48          */
49         public function up()
50         {
51             Schema::create('category_product', function (Blueprint $table) {
52                 $table->increments('id');
53                 $table->string('category_id');
```

```
            $table->string('product_id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('category_product');
    }
}
```

Não esqueça de atualizar o banco com o comando `php artisan migrate`.

Criado um arquivo de Seed

E que tal se inserirmos um usuário inicial, assim quando formos instalar o projeto no servidor bastará rodar uma linha de comando e pronto, estaremos aptos a logar no sistema.

Vá até o arquivo em *database/seeds/DatabaseSeeder.php* e adicione esta classe no fim do arquivo.

```
class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        \App\User::create([
            'name'=>'Erik Figueiredo',
            'email'=>'erik.figueiredo@gmail.com',
            'password'=>bcrypt('123456'),
        ]);
    }

}
```

E dentro do método run() da classe DatabaseSeeder:

```
1 $this->call('UserTableSeeder');
```

Precisa explicar? Ta bom, explico, a classe UserTableSeeder é responsável por inserir um registro usando o `Eloquent` (sim, o `User::create()` é um comando do Eloquent para inserir registros), o DatabaseSeeder::run() carrega e executa a classe, já o comando `DB::table('users')->delete();` remove todos os dados da tabela.

Não se preocupe em entender isso agora, vamos nos aprofundar bem mais para frente.

Para efetivar o registro no banco basta rodar o comando:

```
1 php artisan db:seed
```

Prontinho, simples!

Model, View e Controller

Rotas

As rotas definem o que acontece na sua aplicação quando determinada URL é acessada, existem muitas formas de trabalhar com rotas no Laravel e isso é incrível se você quer liberdade e estabilidade, quem não quer?

Um bom exemplo para começarmos é aquela página inicial do Laravel, sabe, aquela tela que você viu no capítulo anterior. Abra o arquivo em *app/Http/routes.php*, este arquivo guarda todas as rotas da nossa aplicação, ele deve se parecer com este exemplo a seguir, tomei a liberdade de traduzir os comentários.

```
1  <?php
2
3  /*
4  |--------------------------------------------------------------------------
5  | Application Routes
6  |--------------------------------------------------------------------------
7  |
8  | Aqui está onde você pode registra todas as rotas para uma aplicação
9  | É uma brisa. Simplesmente diga ao Laraval a URI que deveria responder
10 | e dar-lhe o controller para chamar quando a URI for solicitada
11 |
12 */
13
14 Route::get('/', function () {
15     return view('welcome');
16 });
```

Estas três linhas simplesmente informam que ao acessar a *"URL raiz"* ('/') o Laravel deve renderizar a view *welcome*, esta view deve estar em *resources/views/welcome.blade.php*, o Laravel é inteligente o suficiente para encontrar um arquivo de view, apenas devemos respeitarmos as seguintes regras.

- Um arquivo de view deve estar dentro de *resources/views*
- um arquivo de view deve terminar com *.php* ou *.blade.php*

Vou explicar mais tarde a diferença entre arquivos *.blade.php* e *.php*, por hora fique a vontade para renomear o arquivo de *welcome.blade.php* para *welcome.php* e ver que não vai mudar nada.

Parâmetros de URL

Vamos brincar mais um pouco? Que tal um **Hello World**? Adicione esta rota no final do arquivo *routes.php*.

```
1  Route::get('/hello', function () {
2      return 'Hello World!';
3  });
```

Agora acesse *http://localhost:8000/hello* e veja seu texto aparecer na tela (não esqueça de iniciar o servidor). O Laravel tem um sistema de rotas muito bem elaborado e você pode fazer muito só com ele.

```
1  Route::get('/hello/{name}', function ($name) {
2      return 'Hello ' . $name . '!';
3  });
```

Agora tente acessar a url *hello/seunomeaqui*, note que você pode adicionar esta nova rota ou substituir a anterior, só que se substituir não vai mais ter acesso a */hello* somente esta nova, substitua. Temos uma url variável com um parâmetro obrigatório, legal, mas agora vamos tornar esse parâmetro opcional, assim a URL */hello* voltará a funcionar.

```
1  Route::get('/hello/{name?}', function ($name = 'World') {
2      return 'Hello ' . $name . '!';
3  });
```

Agora você tem uma rota com parâmetro opcional e um valor padrão caso `$name` não seja enviado.

Retringindo por verbos HTTP

Você notou que até agora usamos `Route::get()` para especificar uma rota? Esse `get()` se refere ao verbo *GET*, então é meio óbvio que temos também um `post()` para o verbo *POST*, correto?

Tente trocar uma das urls que criamos para *POST* e acesse, por exemplo:

```
1  Route::post('/hello/{name?}', function ($name = 'World') {
2      return 'Hello ' . $name . '!';
3  });
```

Você deve receber um erro que, entre outras coisas, informa o erro **MethodNotAllowedHttpException in RouteCollection.php line 201**, isso quer dizer que funcionou, você só pode acessar a url através do método POST, caso contrário receberá um erro de permissão, isso é interessante porque limita o acesso a um recurso apenas para os versos especificados, ou seja, não deixa margem para requisições inesperadas e maliciosas, mas como fazer para liberar o acesso a uma rota em dois verbos HTTP ao mesmo tempo? Criamos duas rotas? Não, para estes momentos usamos o `match()`.

```
Route::match(['get', 'post'], '/hello/{name?}', function ($name = 'World') {
    return 'Hello ' . $name . '!';
});
```

Em vez de passarmos a rota no primeiro parâmetro, passamos no segundo, e no primeiro informamos um array com os verbos que queremos dar acesso.

Prontinho, agora que você entendeu essa parte quero apenas reforçar que você também tem acesso a outros verbos HTTP, como PUT e DELETE, não conhece? Passou da hora de procurar saber mais sobre isso então.

```
Route::put('foo/bar', function () {
    //
});

Route::delete('foo/bar', function () {
    //
});
```

Brincando com Banco de dados

Que tal brincarmos um pouco com banco de dados? Lembra aquele usuário que cadastramos com os seeds do Laravel? Vamos pegar ele do banco e imprimir na tela?

```
Route::get('/all_users', function () {
        return DB::select('SELECT * FROM users WHERE 1=1');
});
```

Note que o resultado foi automaticamente convertido para uma saída json, o Laravel é mágico, adicione um *.json* ao final da rota, para diferenciar do exemplo seguinte. Não se preocupe em entender o DB::select() agora, cada coisa em seu tempo.

Para listar estes dados em um HTML, ou seja, uma lista com um titulo.

```
Route::get('/all_users', function () {
        $users = DB::select('SELECT * FROM users WHERE 1=1');
        return view ('users.aprendendo_rotas', ['users'=>$users]);
});
```

Sempre que acessarmos um arquivo de view no Laravel que deveria estar dentro de um diretório usamos o ponto como caracter de separação, então não esqueça de criar um arquivo *users/aprendendo_rotas.php* dentro de *resources/views*, com o código abaixo.

```
<h1>Users</h1>
<ul>
        <?php foreach ($users as $user): ?>
        <li><?php echo $user->name;?></li>
        <?php endforeach; ?>
</ul>
```

Agora quando carregarmos a url no navegador você deve receber um HTML mais elaborado e com o único usuário cadastrado que temos listado, com o tempo teriamos mais usuários e mais registros a exibir.

Note que a variável `$users` foi compartilhada a partir da rota na view, isso graças ao segundo parâmetro do método `view()`, ele informa o nome da variável e o valor que ela deverá ter, se você quiser que a variável tenha o nome `$data` na view, o método seria criado conforme a seguir.

```
return view ('users.aprendendo_rotas', ['data'=>$users]);
```

E o `foreach()` na view ficaria assim.

```
<?php foreach ($data as $user): ?>
```

Simples na verdade.

Carregando controllers e actions

Mas se você criar uma aplicação desta forma, rapidamente seu `routes.php` ficará gigantesco e dificil de manter, então que tal delegar a tarefa de requisitar e retornar dados para um controller, afinal é pra isso que ele serve.

```
Route::get('/all_users', ['uses'=>'UsersController@allUsers']);
```

Prontinho, agora eu informei que vou usar o controller *UsersController* e a action *allUsers()* para executar a tarefa em questão.

Mas ainda não temos um controller, então se você testar a rota agora vai receber uma mensagem informando que o controller não existe, assim: *Class AppHttpControllersUsersController does not exist.*

Nomeando rotas

Sempre que precisarmos fazer referência a uma rota qualquer, seja em links na view ou redirecionamentos no controller é sempre uma boa ideia usar o nome da rota em vez da URL em sí, este deve ser único e bem pensado, assim se sua rota mudar você ainda mantém o link ativo.

Para nomear uma rota apenas informe o parâmetro `as` no array de configuração.

```
Route::get('/all_users', ['uses'=>'UsersController@allUsers', 'as'=>'users.all_u\
sers']);
```

Neste caso o ponto não tem qualquer significado maior, apenas separa os nomes.

Criando um controller

Agora que você já tem sua rota vamos criar um controller, acesse *app/Http/Controllers* e crie um arquivo chamado *UsersController.php* e crie uma classe padrão do PHP, como já conhecemos:

```
<?php

namespace App\Http\Controllers;

class UsersController
{
}
```

Todo controller deve hedar Controller, que por sua vez herda de BaseController, o resultado segue a baixo.

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use class UsersController extends Controller;
{
}
```

E por fim nosso action *allUsers()*:

```php
<?php

namespace App\Http\Controllers;

class UsersController extends Controller
{
        public function allUsers()
    {
        $users = \DB::select('SELECT * FROM users WHERE 1=1');
                return view ('users.aprendendo_rotas', ['users'=>$users]);
    }
}
```

Já separamos melhor as camadas, se acessar agora a rota */all_users* vai receber o mesmo resultado que antes, mas com uma estrutura mair organizada e passiva de crescer sem prejudicar sua organização.

Criando um controller com artisan

Mas o Laravel poderia ter criado este controller pra gente e ainda preparado uma estrutura de actions muito interessante e que vamos usar mais pra frente, apague o UsersController e vamos fazer novamente.

```
php artisan make:controller UsersController
```

Se, por qualquer motivo, você não quiser que o controller venha com todos este métodos pode usar o parametro *–plain* e criar um controller vazio, faça alguns testes, o comando completo ficaria assim:

```
php artisan make:controller UsersController --plain
```

Adicione o método *allUsers()* no nosso novo controller.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

```

```
10  class UsersController extends Controller
11  {
12
13          public function allUsers()
14      {
15          $users = \DB::select('SELECT * FROM users WHERE 1=1');
16                  return view ('users.aprendendo_rotas', ['users'=>$users]);
17      }
18
19      /**
20       * Mostra uma lista de registros
21       *
22       * @return Response
23       */
24      public function index()
25      {
26          //
27      }
28
29      /**
30       * Exibe um formulário de criação de registro
31       *
32       * @return Response
33       */
34      public function create()
35      {
36          //
37      }
38
39      /**
40       * Armazena um novo registro
41       *
42       * @return Response
43       */
44      public function store()
45      {
46          //
47      }
48
49      /**
50       * Exibe um registro específico
51       *
```

```
52          * @param  int  $id
53          * @return Response
54          */
55         public function show($id)
56         {
57             //
58         }
59
60         /**
61          * Exibe um formulário de edição de registros
62          *
63          * @param  int  $id
64          * @return Response
65          */
66         public function edit($id)
67         {
68             //
69         }
70
71         /**
72          * Atualiza um registro específico
73          *
74          * @param  int  $id
75          * @return Response
76          */
77         public function update($id)
78         {
79             //
80         }
81
82         /**
83          * Remove um registro específico
84          *
85          * @param  int  $id
86          * @return Response
87          */
88         public function destroy($id)
89         {
90             //
91         }
92 }
```

Consultando o banco de dados

Você deve estar muito interessado naquela classe *DB* que usamos para buscar dados do banco, tenho certeza que sim, pois bem, vamos ver a fundo o que podemos fazer com ela?

O objeto DB é a forma mais simples que temos para executar queries Sql no Laravel 5, veja a seguir um exemplo com proteção contra injeção de dados.

```
1  $users = \DB::select('SELECT * FROM users WHERE id=:id', ['id'=>1]);
```

Este código deve retornar o usuário de *id* 1 (provavelmente o que você já tem cadastrado), o que ele fez foi substituir o valor de *:id* pelo que informei no *array*, isso evita o processo de *Sql Injection* usando o *PDO* para escrever o valor de forma segura.

Você também poderia fazer desta forma:

```
1  $users = \DB::select('SELECT * FROM users WHERE id=?', [1]);
```

Agora ele vai trocar o *?* por 1, o primeiro exemplo é chamado de named binding, ou binding nomeado é o que eu prefiro usar no meu dia a dia (quando escrevo Sql, ou seja, quase nunca em uma aplicação PHP).

Nós usamos o método *select()* para consultar os dados, mas ainda temos o *insert()*, o *update()*, o *delete()* e o *statement()*.

```
1  DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
2
3  $affected = DB::update('update users set votes = 100 where name = ?', ['John']);
4
5  $deleted = DB::delete('delete from users');
6
7  DB::statement('drop table users');
```

Os métodos são auto-explicativos, não precisa de muitos detalhes pra você entender, o `statement()` serve para executar queries diversas como remover uma tabela (`drop table`), criar um banco de dados (`criate database`), enfim.

Ainda temos *Ouvintes de consulta*, *transações* e *multiplas conexões* ao banco de dados na documentação, o objeto `DB` do Laravel é bem completo e abstrai muito bem o PDO. Para ir mais a fundo consulte http://laravel.com/docs/5.1/database[14].

[14] http://laravel.com/docs/5.1/database

Query builder

Eu tenho o costume de passar a maior parte da responsabilidade possível para o PHP em se tratando de banco de dados, já disse isso antes, e uso muito *Query Builders*, com eles você constrói sql usando métodos e o Laravel se vira pra gerar o comando em *string*, já sabendo que existem muitas diferenças na escrita de Sql entre diferentes bancos de dados, acredito ser uma prática muito aceitável.

Lembra desta linha:

```
1  $users = \DB::select('SELECT * FROM users WHERE id=:id', ['id'=>1]);
```

Você consegue o mesmo resultado com o código a baixo, mas agora o Laravel se encarrega de criar o sql correto para o seu banco de dados, você até pode trocar o banco quando quiser, vai continuar funcionando.

```
1  $user = DB::table('users')->where('id', 1)->get();
```

Em vez do *get()* eu poderia usar o *first()* para trazer o primeiro resultado, ou o *chunk(100, Closure)* para trazer partes de uma consulta por vez (100 registros de cada vez, por exemplo), ou o *lists()* para gerar listas, ou o *count()* (e os outros agregadores, como *max()*, *avg()*, enfim).

Veja como ficaria nosso exemplo anterior usando query builder.

```
1  DB::table('users')->insert(
2          'id' => 1,
3          'name' => 'Dayle'
4  );
5
6  $affected = DB::table('users')
7      ->where('name', 'John')
8      ->update(['votes' => 100]);
9
10 $deleted = DB::table('users')->delete();
11
12 DB::statement('drop table users');
```

Ainda temos vários outros exemplos úteis que poderíamos trabalhar aqui, mas vamos fazer isso durante o desenvolvimento do nosso projeto principal, vamos nos focar.

Criando um model

A minha forma favorita de trabalhar com banco de dados é com models, o resultado é um controller mais limpo ao tempo que a liberdade é muito grande, já que toda model retorna um query builder para usarmos, por exemplo, ambas as linhas abaixo fazem a mesma coisa, mas a segunda foi escrita usando a model `User` que já vem no Laravel.

```
DB::table('users')->delete();
\App\User::delete();
```

Eu agora tenho um controle maior da minha aplicação, já que todas as configurações da tabela `users` está dentro do model `App\User`.

No Laravel 5 os models são criados na raiz do diretório *app*, nada impede de você colocar seus models em um diretório personalizado, talvez em *app/Http/Models* ou até *app/Models*, desde que respeite o namespace da classe, eu vou manter no formato padrão, ou seja, no diretório `app`.

Vamos criar nosso segundo model! Opa, segundo? Sim, o Laravel já traz o model *User* criado, menos trabalho pra gente, vamos criar outro, que tal o *Product*, então crie uma classe em *app/Product.php*, ela deve ficar assim:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    //
}
```

Só que mais uma vez poderiamos ter usado o Artisan.

```
php artisan make:model Product
```

O resultado seria exatamente o mesmo.

A primeira coisa a se fazer é informar qual tabela do banco de dados este model deve ser responsável (sim, teremos um model por tabela), para fazer isso usamos o atributo protegido *$table*, sugestivo, não.

```
protected $table = 'products';
```

Também podemos informar quais campos podem ser criados em massa, ou seja, atribuidos de uma vez, são dois atributos agora, o *$fillable* para permitir as inserções e o *guarded* para proibir.

```php
/**
 * Permitir alterações em massa
 */
protected $fillable = ['title', 'body', 'value', 'qtd'];

/**
 * Proibir alterações em massa
 */
protected $guarded = ['url'];
```

Eu mostrei o atributo `$guarded` apenas para ilustrar, na verdade o `url` também deve ficar dentro do `$fillable`, não esqueça de fazer essa alteração.

Ainda temos os atributos *$timestamps* e *$primaryKey*, o primeiro indica (com true ou false) se você vai usar os campos `created_at` e `updated_at`, o padrão é *true*, caso você não tenha criado estes campos vai precisar informar *false*. O segundo atributo (*$primaryKey*) vai informar qual a chave primária da tabela, o padrão é *id* e não é necessário informa-lo caso esta seja a sua chave primária.

Meu model completo ficou assim.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $table = 'products';

    protected $fillable = ['title', 'body', 'value', 'qtd', 'url'];
}
```

Entidades do Eloquent

Agora que nosso model está pronto, podemos usar o Eloquent (ORM do Laravel) para fazer as consultas de uma forma muito mais limpa e organizada, ele vai usar a classe de model que criamos e isso já facilita muito. O conceito de ORM trabalha com entidades representado cada registro em vez de simples arrays, estas entidades são objetos comuns e que podem ser manipulados através do arquivo de model.

Vamos usar muito as entidades e models, não se preocupe em entender tudo agora.

Imagine a seguinte classe (não precisa criar, só conhecer a estrutura).

```
class User
{
        public $id;
        public $name;
        public $email;
        public $password;
        public $created_at;
        public $updated_at;
}
```

Essa classe é uma entidade e cada atributo representa um campo na tabela, quando recuperamos vários dados do banco teremos um array com várias entidades dentro. Simples assim.

Este é um exemplo de consulta e impressão na tela.

```
$user = \App\User::find(1);

echo '<h1>' . $user->title . '</h1>';
echo $user->id . '<br>';
echo $user->email;
```

Vamos avançar um pouco as coisas e começar a criar nosso painel de administração da loja.

CRUD

Crie o controller *ProductsController* com o artisan sem `--plain`, o comando completo:

```
1  php artisan make:controller ProductsController
```

O controller rescem criado tem 7 métodos, cada um com sua responsabilidade específica, agora vamos criar 7 rotas, uma para cada método, mas antes faça as contas, como teremos um CRUD de categorias e outro de usuários então são 21 rotas, 3 controllers com 7 métodos cada, agora imagine 9 ou 10 controllers, imagine 15, desanimou? Calma, calma, não criemos pânico, o Laravel já implementa uma rota que cuida dessa estrutura pra gente, veja a mágica, vá até o *routes.php* e insira um novo roteamento usando o comando abaixo.

```
1  Route::resource('products', 'ProductsController');
```

Com apenas o método `resource()` temos acesso a 7 URLs padrão no nosso crud, todas nomeadas e específicas a seus verbos HTTP.

verbo	url	action	Nome da rota
GET	/products	index	products.index
GET	/products/create	create	products.create
POST	/products	store	products.store
GET	/products/{id}	show	products.show
GET	/products/{id}/edit	edit	products.edit
PUT/PATCH	/products/{id}	update	products.update
DELETE	/products/{id}	destroy	products.destroy

Note a última coluna com os nomes da rota, vamos usar muito isso.

Estamos trabalhando com uma estrutura base para um servidor RESTful, mas também é excelente para o que precisamos fazer e é fantástico como o Laravel simplifica o nosso trabalho neste ponto, esse formato é mais organizado, as actions acabam seguindo o Simple Principle (princípio da responsabilidade única) do S.O.L.I.D., além de facilitar a prática de Calisthenics Object, que dita um nível de identação em cada método, outro ponto a favor é que os nossos actions estão protegidos com acesso a verbos HTTP específicos, além da proteção CSRF padrão do Laravel 5.

Prontinho, vamos criar as views!

Criando views

Vá até o diretórios *resources/views* e crie os seguintes arquivos dentro de um novo diretório *products* (crie caso não existam).

- index.blade.php
- create.blade.php
- show.blade.php
- edit.blade.php

Listando produtos

Vamos fazer a primeira consulta com o Eloquent no nosso controller ProductsController?

Na action *index()* adicione estas duas linhas.

```
$products = Product::all();
return view ('products.index', ['products'=>$products]);
```

A primeira vai trazer todos os resultados da tabela *users* enquanto a segunda vai informar qual view usar e enviar os dregistros para lá, limpo, simples e bonito.

A view em *products/index.blade.php* deve ficar assim, por enquanto.

```
<h1>Usuários</h1>

<table>
        <thead>
                <tr>
                        <th>id</th>
                        <th>title</th>
                        <th>actions</th>
                </tr>
        </thead>
        <tbody>
        @foreach ($products as $k=>$product)
                <tr>
                        <th>{{ $product->id }}</th>
                        <th>{{ $product->name }}</th>
                        <th>
                                <a href="{{ route('users.show', ['id'=>$product->id]) }}">view</a>
```

```
                                <a href="{{ route('users.edit', ['id'=>$product->id]) }}">edit</a>
                                <a href="{{ route('users.destroy', ['id'=>$product->id]) }}">remov
                            </th>
                    </tr>
            @endforeach
            </tbody>
</table>
```

Note que nenhuma tag `<?php` foi usada, em vez disso eu usei marcações do Blade. O Laravel vem com um *template engine* que facilita a escrita de arquivos de view, ele se chama Blade. As duas marcações que usamos:

- {{ string }} - É o mesmo que `echo` do PHP
- {{ @foreach }} - É o mesmo que `foreach()` do PHP
- {{ @endforeach }} - informa aonde o `@foreach` termina

Além disso eu usei um método `route()`, aqui já temos código PHP sem Blade, este *helper* traduz os nomes das rotas em urls, também temos acesso ao `route()` no controller, através do método `redirect()`, como veremos mais a frente.

Cadastrando um produto

Para inserir um registro vamos precisar de duas actions, a *create()* para exibir o formulário de cadastro e a *store()* para realmente incluir no banco de dados. Lembre-se que a create() será acessada via GET e a store() via POST, vamos criar o código?

```
public function create()
{
    return view ('products.create');
}

public function store(Request $request)
{
    dd($request->all());
}
```

A novidade agora é o objeto *Request* que é instanciado na variável *$request* automaticamente pelo IoC do Laravel sem que você ao menos precise saber o que é um IoC, através dele podemos ter acesso a todos os dados enviados pela requisição como as variáveis GET e POST, por exemplo. O `dd()` significa *dump and die* e executa uma espécie de *var_dump()* seguido de um *die()*, neste momento é interessante sabermos como estes dados são recebidos no controler para em seguida fazer algo com eles, neste caso, persistir no banco.

Veja alguns exemplos para lidar com dados GET e POST.

```php
$name = $request->input('name'); // campo name enviado por qualquer verbo HTTP
$name = $request->input('name', 'Sally'); // agora com como valor padrão 'Sally'\
 caso não seja encontrado

/**
 * Verificando se um campo foi enviado
 */
if ($request->has('name')) {
        /*... foi enviado ...*/
}

$input = $request->all(); // retorna todos os campos em qualquer verbo HTTP
```

Agora que você está afiado que tal construirmos um formulário para enviar estes dados, a view em *products/create.blade.php* fica assim:

```php
<h1>Cadastrar usuário</h1>

<form action="{{ route('products.store') }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        Title: <input type="text" name="title"><br>
        Description: <textarea name="body"></textarea><br>
        Value: <input type="text" name="value"><br>
        Quantity: <input type="number" name="qtd"><br>
        Url: <input type="text" name="url"><br>
        <input type="submit">
</form>
```

Viu o campo *_token*, o Laravel já vem nativamente com proteção CSRF ativada, você só precisa informar o valor do campo com o helper *csrf_token()* que vai retornar o hash a ser usado. CSRF é um acrônimo de Cross Site Request Forgery, ou Falsificação de Solicitações entre sites e consiste em um tipo de ataque focado em enviar falsas requisições, ele pode ser bloqueado com um hash aleatório que deve ser enviado em todas as requisições diferentes de GET.

Ao preencher e enviar o formulário você deve se deparar com os dados impressos na tela, tente remover o campo *_token* e enviar, você vai receber um erro. Com os dados impressos na tela, vamos persistílos no banco. Troque o conteúdo de *store()* por:

```php
Product::create($request->all());
return redirect()->route('products.index');
```

Usei nosso model para fazer um *Mass Assignment*, ou seja, enviar os valores todos de uma vez para o Eloquent, esta não é a única forma de fazer isso, para quem já trabalhou com outros ORMs vai se sentir mais em casa com este exemplo a seguir, embora o anterior traga óbvios benefícios.

```
$product = new Product;

$product->title = $request->input('title');
$product->value = $request->input('value');
$product->qtd = $request->input('qtd');
$product->url = $request->input('url');
$product->body = $request->input('body');

$product->save();
return redirect()->route('products.index');
```

O *Product::create()* faz exatamente a mesma coisa que este último, mas internamente o model verifica que dados vai aceitar (com base no atributo `$fillable`) e quais tratamentos precisa fazer (nenhum por enquanto, posteriormente vamos tratar o campo *url*).

Note também que eu usei um tal de *return redirect()*, este cara é responsável por fazer os redirecionamentos (óbvio) que precisamos durante a execução da aplicação. o *route()* após o redirect envia a requisição para uma rota nomeada, como usamos o 'Route::resource()' já temos os nomes das rotas criado por ele (veja a tabela que preparei no começo do capítulo), nada mais justo eu usar. Veja outras formas de usar o *redirec()* a seguir.

```
return redirect('home/dashboard'); // redireciona para localhost:8000/home/dashb\
oard
return back(); // volta para a página anterior
return redirect()->action('UsersController@index'); //redireciona para o control\
ler e action especificados
```

Retornando um produto

Este é simples, já até usamos algo anteriormente, coloque isso dentro do action show().

```
$product = Product::find($id);
return view ('products.show', ['product'=>$product]);
```

Estamos trazendo o usuário com *$id* informado na URL, é um query simples, mas e se quisermos algo mais sofisticado, com multiplos resultados (um Product::all() como vimos antes), com ordenação personalizada, limitado a 10 resultados, várias condições, enfim, mais personalizado, como fariamos?

```
$flights = \App\Flight::where('active', 1)
   ->where('destination', 'San Diego')
   ->orderBy('name', 'desc')
   ->take(10)
   ->get(); //para trazer apenas um usuário podemos usar o ->first(); ao invés d\
o ->get();
```

Mas foi só um exemplo, nem temos um model *Flight*. E aqui a nossa view:

```
<h1 class="page-header">{{ $data->title }}</h1>

<ul>
        <li>value: {{ $data->value }}</li>
        <li>qtd: {{ $data->qtd }}</li>
        <li>url: {{ $data->url }}</li>
        <li>cadastro: {{ $data->created_at }}</li>
        <li>atualização: {{ $data->updated_at }}</li>
</ul>
<hr>
<p>Description</p>
{{ $data->body }}
```

Editando o usuário

Para a edição vamos usar um método chamado *update()*, o processo é parecido com o que já fizemos antes no cadastro, ou seja, duas actions e uma view, desta vez é a *edit()* para exibir o formulário e a *update()* para atualizar os dados.

Seguem as actions.

```
public function edit($id)
{
    $user = User::find($id);
    return view ('users.edit', ['user'=>$user]);
}

public function update(Request $request, $id)
{
    $user = User::find($id);
    $user->update($request->all());
    return redirect()->route('users.index');
}
```

Muito parecido com o que fizemos antes, mas agora buscamos um registro para em seguida atualizar os dados.

Você também pode passar campo por campo, assim:

```
$product = Product::find(1);

$product->title = $request->input('title');
$product->value = $request->input('value');
$product->qtd = $request->input('qtd');
$product->url = $request->input('url');
$product->body = $request->input('body');

$product->save();
return redirect()->route('users.index');
```

A nossa view.

```
<h1>Editando {{ $user->name }}</h1>

<form action="{{ route('products.update', ['id'=>$data->id]) }}"method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        <input type="hidden" name="_method" value="PUT">
        Title: <input type="text" name="title" value="{{ $data->title }}"><br>
        Description: <textarea name="body">{{ $data->body }}</textarea><br>
        Value: <input type="number" name="value" value="{{ $data->value }}"><br>
        Quantity: <input type="number" name="qtd" value="{{ $data->qtd }}"><br>
        Url: <input type="text" name="url" value="{{ $data->url }}"><br>
        <input type="submit">
</form>
```

Se você consultou a tabela da rota resources já sabe que uma requisição POST não será bem vinda ao enviar o formulário, precisamos de um PUT, mas formulários HTTP não dão suportea verbos diferentes de POST e GET, novamente o Larave surpreende, de uma atenção ao campo *_method* do nosso formulário, a solução é usar este campo oculto com o valor do verbo HTTP que quer usar, neste caso o PUT, assim o Laravel refaz a requisição da forma correta.

Adicionalmente podemos criar um mutator para o campo url? A ideia é que o valor seja baseado no campo *title* sempre que o *url* for deixado em branco, isso deve ser feito no model Product.

```
public function setUrlAttribute($value)
{
    if ($value=='')
        $value = $this->attributes['title'];

    $this->attributes['url'] = str_slug($value);
}
```

Um mutator é um método que criamos na model para manipular os dados do campo.

Note que eu usei um helper do Laravel, o `str_slug()`, para saber mais sobre os helpers, veja este link da documentação: http://laravel.com/docs/5.1/helpers[15].

Por falar em mutators, que tal já criarmos o de senha, valores em branco também geral hash, precisamos ensinar o Laravel ignorar campos de senha em branco, este método deve ficar no seu modelUser.

```
public function setPasswordAttribute($value)
{
    if ($value=='')
        unset($this->attributes['password']);
    else
        $this->attributes['password'] = bcrypt($value);
}
```

Também usei um helper aqui, o bcrypt, agora nosso model gera o Hash automáticamente.

Veja como ficaram nossos models.

```
<?php

namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements AuthenticatableContract, CanResetPasswordCon\
tract
{
```

[15] http://laravel.com/docs/5.1/helpers

```
14      use Authenticatable, CanResetPassword;
15
16      /**
17       * The database table used by the model.
18       *
19       * @var string
20       */
21      protected $table = 'users';
22
23      /**
24       * The attributes that are mass assignable.
25       *
26       * @var array
27       */
28      protected $fillable = ['name', 'email', 'password'];
29
30      public function setPasswordAttribute($value)
31      {
32          if ($value=='')
33              unset($this->attributes['password']);
34          else
35              $this->attributes['password'] = bcrypt($value);
36      }
37  }
38
39  <?php
40
41  namespace App;
42
43  use Illuminate\Database\Eloquent\Model;
44
45  class Product extends Model
46  {
47      protected $table = 'products';
48
49      /**
50       * Permitir alterações em massa
51       */
52      protected $fillable = ['title', 'body', 'value', 'qtd', 'url'];
53
54      public function setUrlAttribute($value)
55      {
```

```
56          if ($value=='')
57              $value = $this->attributes['title'];
58
59          $this->attributes['url'] = str_slug($value);
60      }
61  }
```

Mutators tem um padrão para os seus nomes, se você não seguir isso o Eloquent vai ignora-los. O padrão é a palavra set seguida do nome do atributo que vamos alterar em CamelCase e encerrado com a palavra Attribute, por exemplo, `setPasswordAttribute()` e `setUrlAttribute()`.

Removendo o usuário

Última etapa do nosso CRUD. Remover um registro é relativamente simples, no action *destroy()* coloque este código.

```
1  $product = Product::find($id);
2  $product->delete();
3  return redirect()->route('users.index');
```

Não sei se você notou, mas não é possível acessar a action *destroy()* no momento, já que o link **remove** da action *index()*‘faz uma requisição GET (o padrão da tag *

```
1  <form action="{{ route('users.update', ['id'=>$user->id]) }}" class="form" metho\
2  d="POST" style="display:inline-block">
3          <input type="hidden" name="_token" value="{{ csrf_token() }}">
4          <input type="hidden" name="_method" value="DELETE">
5          <input type="submit" value="remove">
6  </form>
```

Agora podemos remover o nosso registro, note que a proteção CSRF impossibilita a remoção dos dados a partir de fontes não autorizadas (sem acesso ao token CSRF).

Abstraindo o CRUD

Agora que já temos o primeiro CRUD, vamos criar os demais, mas eu simplesmente odeio repetição, mais que isso, odeio perder tempo, se você também pensa assim saiba que tem uma jogada muito legal aqui, vamos abstrair esse CRUD, duplique o *ProductsController* chamando o novo arquivo de *CrudController*.

Vou remover todos os comentários por conta do espaço aqui, transformar a classe em abstrata e criar um novo método para retornar a model. Veja todas as alterações.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

abstract class CrudController extends Controller
{

        protected $model_instance;

    public function index()
    {
        $data = $this->getModel()->all();
        return view ($this->path.'.index', ['data'=>$data]);
    }

    public function create()
    {
        return view ($this->path.'.create');
    }

    public function store(Request $request)
    {
        $this->getModel()->create($request->all());
        return redirect()->route($this->path.'.index');
    }

    public function show($id)
    {
        $data = $this->getModel()->find($id);
        return view ($this->path.'.show', ['data'=>$data]);
    }

    public function edit($id)
    {
        $data = $this->getModel()->find($id);
        return view ($this->path.'.edit', ['data'=>$data]);
    }
```

```
43
44      public function update(Request $request, $id)
45      {
46          $data = $this->getModel()->find($id);
47          $data->update($request->all());
48          return redirect()->route($this->path.'.index');
49      }
50
51      public function destroy($id)
52      {
53          $data = $this->getModel()->find($id);
54          $data->delete();
55          return redirect()->route($this->path.'.index');
56      }
57
58      protected function getModel()
59      {
60              if ($this->model_instance === null)
61                      $this->model_instance = new $this->model;
62
63              return $this->model_instance;
64      }
65  }
```

Agora volte ao *ProductsController* e remova todos os métodos e adicione dois novos atributos

- $model - O namespace do model.
- $path - o diretório aonde as views do crud vão ficar e a base do nome da rota.

Você também pode separar o `$path` em dois atributos distindos, `$view_path` e `$route`, assim ganha a liberdade de alterar um independente do outro, achei desnecessário no nosso caso, fique a vontade para decidir, mas não esqueça das devidas alterações no *CrudController*.

Depois troque o extends *Controller* do *ProductsController* por *CrudController*.

O *ProductsController* completo:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\CrudController;

class ProductsController extends CrudController
{
    protected $model = '\App\Products';
    protected $path = 'products';
}
```

Pronto, agora conseguimos simplificar nosso controller, muito melhor, você ainda deve ver um erro nas views, já que troquei as variáveis `$products` por `$data`, é só fazer essa correção. Vou aproveitar e criar o CRUD do *ProductsController*.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\CrudController;

class UsersController extends CrudController
{

    protected $model = '\App\User';
    protected $path = 'users';

}
```

Agora só precisamos dos arquivos de view do `UsersController`, veja como eles devem ficar.

```
//users/create.blade.php
<h1 class="page-header">{{ $data->name }}</h1>

<ul>
        <li>email: {{ $data->email }}</li>
        <li>cadastro: {{ $data->created_at }}</li>
        <li>atualização: {{ $data->updated_at }}</li>
</ul>

//users/edit.blade.php
<h1 class="page-header">Editando {{ $data->name }}</h1>

<form action="{{ route('admin.users.update', ['id'=>$data->id]) }}" class="form"\
 method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        <input type="hidden" name="_method" value="PUT">
        Name: <input type="text" name="name" class="form-control" value="{{ $data->name\
 }}"><br>
        Email: <input type="email" name="email" class="form-control" value="{{ $data->e\
mail }}"><br>
        Password: <input type="password" name="password" class="form-control" value="">\
<br>
        <input type="submit" class="btn btn-primary">
</form>


//users/index.blade.php
<h1 class="page-header">
        Usuários
        <small><a href="{{ route('admin.users.create') }}" class="btn btn-success btn-x\
s">novo</a></small>
</h1>

<table class="table table-hover table-striped">
        <thead>
                <tr>
                        <th>id</th>
                        <th>name</th>
                        <th>mail</th>
                        <th class="text-right">actions</th>
                </tr>
        </thead>
```

```
43          <tbody>
44          @foreach ($data as $k=>$v)
45                  <tr>
46                          <td>{{ $k+1 }}</td>
47                          <td>{{ $v->name }}</td>
48                          <td>{{ $v->email }}</td>
49                          <td class="text-right">
50                                  <a href="{{ route('admin.users.show', ['id'=>$v->id]) }}" class="b
51  mary btn-xs">view</a>
52                                  <a href="{{ route('admin.users.edit', ['id'=>$v->id]) }}" class="b
53  ault btn-xs">edit</a>
54                                  <form action="{{ route('admin.users.update', ['id'=>$v->id]) }}" c
55  " method="POST" style="display:inline-block">
56                                          <input type="hidden" name="_token" value="{{ csrf_token()
57                                          <input type="hidden" name="_method" value="DELETE">
58                                          <input type="submit" value="remove" class="btn btn-xs btn-
59                                  </form>
60                          </td>
61                  </tr>
62          @endforeach
63          </tbody>
64  </table>
65
66
67  //users/show.blade.php
68  <h1 class="page-header">{{ $data->name }}</h1>
69
70  <ul>
71          <li>email: {{ $data->email }}</li>
72          <li>cadastro: {{ $data->created_at }}</li>
73          <li>atualização: {{ $data->updated_at }}</li>
74  </ul>
```

No próximo capítulo vamos incrementar mais nossa aplicação validando os dados e também criando o CRUD de categorias para relacionarmos com produtos.

Validações

Uma parte essencial do processo de desenvolvimento de qualquer aplicação para qualquer que seja o fim.

A camada de validação ajuda o seu usuário a enviar dados da forma esperada enquanto mantem a segurança. Existem diversas formas de validar dados com Laravel 5, uma delas é com o `$this->validate` direto no controller, o método recebe dois parâmetros, o primeiro é o objeto Request (que usamos para pegar os dados do formulário) e o segundo é um array com as regras de validação.

```
$this->validate($request,
        [
                'title' => 'required|min:3',
                'body' => 'required',
                'value' => 'required|numeric',
                'qtd' => 'required|numeric',
        ]
);
```

O action a seguir valida os dados antes de salvar um produto, se colocado no `ProductsController` ele vai substituir o `store()` do `CrudController`, esta é uma forma de escrever seus actions do crud com código personalizado para a cituação, já que o action `CrudController@store` deixa de ser executado para dar lugar ao novo `ProductsController@store`.

```
public function store(Request $request)
{
        $this->validate($request,
                [
                        'title' => 'required|min:3',
                        'body' => 'required',
                        'value' => 'required|numeric',
                        'qtd' => 'required|numeric',
                ]
        );

    $this->getModel()->create($request->all());
    return redirect()->route($this->path.'.index');
}
```

Para ver todas as validações disponíveis veja este link da documentação: http://laravel.com/docs/5.1/validation#avail validation-rules[16].

Com isso nossa view também já tem acesso as mensagens de erros, vá até `products/create.blade.php` e adicione o seguinte:

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

O arquivo completo deve ficar assim:

```
<h1>Cadastrar</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<form action="{{ route('products.store') }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        Title: <input type="text" name="title"><br>
        Description: <textarea name="body"></textarea><br>
        Value: <input type="number" name="value"><br>
        Quantity: <input type="number" name="qtd"><br>
        Url: <input type="text" name="url"><br>
        <input type="submit">
</form>
```

Tente enviar o formulário vazio ou o campo title com 2 ou menos caracteres e você receberá mensagens de erro e o registro não será salvo. Claro, as mensagens estão em inglês e talvez não te agradem, que tal substituir? Para isso incluimos um terceiro array nas validações.

[16] http://laravel.com/docs/5.1/validation#available-validation-rules

```
$this->validate($request,
        [
                'title' => 'required|min:3',
                'body' => 'required',
                'value' => 'required|numeric',
                'qtd' => 'required|numeric',
        ],
        [
                'required' => ':attribute não deve ficar vazio.',
                'min' => ':attribute deve ter mais de :min caracteres.',
                'numeric' => ':attribute deve ser um número.'
        ]
);
```

Com isso todos os itens com validação `required`, `min` e `numeric` receberão estas mensagens, se você quiser algo mais específico basta usar `campo.regra` na chave do array, exemplo:

```
$this->validate($request,
        [
                'title' => 'required|min:3',
                'body' => 'required',
                'value' => 'required|numeric',
                'qtd' => 'required|numeric',
        ],
        [
                'required' => ':attribute não deve ficar vazio.',
                'title.required' => 'O título é obrigatório', //aqui o campo title tem uma men\
sagem personalizada
                'min' => ':attribute deve ter mais de :min caracteres.',
                'numeric' => ':attribute deve ser um número.'
        ]
);
```

Por fim vamos resolver um bug que apareceu, quando algum dado não valida o formulário tem todos os dados apagados, para isso basta usarmos o `old('campo')` nos capos do formulário, assim:

```
<form action="{{ route('products.store') }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        Title: <input type="text" name="title" value="{{ old('title') }}"><br>
        Description: <textarea name="body">{{ old('body') }}</textarea><br>
        Value: <input type="text" name="value" value="{{ old('value') }}"><br>
        Quantity: <input type="number" name="qtd" value="{{ old('qtd') }}"><br>
        Url: <input type="text" name="url" value="{{ old('url') }}"><br>
        <input type="submit">
</form>
```

Agora sim, o formulário de criação de produtos está perfeito no que diz respeito a funcionalidade, existem outras formas de conseguir o mesmo efeito, outra solução é usar o `Validator::make()`.

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
                //em caso de falha
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }
```

```
30
31          // em caso de sucesso (armazena os dados, por exemplo)
32      }
33  }
```

Agora que tal copiar isso para a action `edit()`? Melhor não, replicar código é horrível, podemos abstrair criando um atributo em cada Controller com o array de validação. Este seria o novo `store()` do CrudController:

```
1   public function store(Request $request)
2   {
3           $this->validate($request, $this->rules,
4                   [
5                           'required' => ':attribute não deve ficar vazio.',
6                           'title.required' => 'O título é obrigatório',
7                           'min' => ':attribute deve ter mais de :min caracteres.',
8                           'numeric' => ':attribute deve ser um número.'
9                   ]
10          );
11
12      $this->getModel()->create($request->all());
13      return redirect()->route($this->path.'.index');
14  }
```

E o `ProductsController`:

```
1   <?php
2
3   namespace App\Http\Controllers;
4
5   use Illuminate\Http\Request;
6
7   use App\Http\Requests;
8   use App\Http\Controllers\CrudController;
9   use Illuminate\Contracts\Validation\Validator;
10
11  class ProductsController extends CrudController
12  {
13      protected $model = '\App\Product';
14      protected $path = 'products';
15      protected $rules = [
16                  'title' => 'required|min:3',
```

```
17                  'body' => 'required',
18                  'value' => 'required|numeric',
19                  'qtd' => 'required|numeric',
20          ];
21
22  }
```

Também não quero que o atributo `$rules` seja obrigatório, então vou adicionar um `$rules` com array vazio no `CrudController`:

```
1   <?php
2
3   namespace App\Http\Controllers;
4
5   use Illuminate\Http\Request;
6
7   use App\Http\Requests;
8   use App\Http\Controllers\Controller;
9   use App\User;
10
11  abstract class CrudController extends Controller
12  {
13
14          protected $model_instance;
15      protected $rules = [];
16
17  //resto do controller
```

O problema desta abordagem, embora funcione e por isso ninguém pode dizer que está errado, é que quebramos algumas regras de boas práticas, como por exemplo o MVC, aonde lógica tem que ficar no model ou pelo menos fora do controller, ou de S.O.L.I.D. aonde um objeto tem que ter um único motivo de existir e a do controller é controlar (não diga!) a execução de outras classes e não ditar regras, ou o próprio propósito das práticas modernas e orientação a objetos que é a independência do código a nível de classe.

Então vamos remover estas regras dai e criar um Form Request, mas vou facilitar, alias, o Laravel vai facilitar, vamos usar o Artisan:

```
1   php artisan make:request ProductsRequest
```

Com isso criamos uma classe ProductsRequest em `app/Http/Requests/ProductsRequest.php` que deve ser exatamente esta:

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class ProductsRequest extends Request
{
    /**
     * Determina se o usuário está autorizado a fazer esta requisição.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Obtem as regras de validação para aplicar a requisição.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

O método `authorize()` diz se o usuário deve ou não ter acesso a requisição, como está ninguém terá acesso, então mude o retorno para `true`, o método `rules()` vai retornar as regras de validação, e por fim vamos criar um método `messages()` com as traduções, nosso novo request deverá ficar assim:

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

class ProductsRequest extends Request
{

    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'title' => 'required|min:3',
            'body' => 'required',
            'value' => 'required|numeric',
            'qtd' => 'required|numeric',
        ];
    }

    public function messages()
    {
        return [
            'required' => ':attribute não deve ficar vazio.',
            'title.required' => 'O título é obrigatório',
            'min' => ':attribute deve ter mais de :min caracteres.',
            'numeric' => ':attribute deve ser um número.'
        ];
    }
}
```

Note que eu removi os comentários. Para aplicar no controller:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\ProductsRequest; // nosso request
use App\Http\Controllers\Controller;
use App\User;

abstract class CrudController extends Controller
{

        //restante da classe

    public function store(ProductsRequest $request)
    {
        $this->getModel()->create($request->all());
        return redirect()->route($this->path.'.index');
    }
    //restante da classe
}
```

Agora fica muito mais limpo e o melhor, unificado, mas esse código não pode ficar assim já que ele também vai usar estas regras de validação para outros controllers, como por exemplo o `UsersController` que deveria ter o seu `UsersRequest` com suas próprias regras de validação, então, mantenha o `CrudController` original (com o `Request` padrão) e vamos brincar abstraindo o `ProductsRequest`. Vamos criar uma nova classe abstrata no mesmo diretório do `ProductsRequest`, por falta de nome melhor vou chamar de `AbstractRequest`.

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\Request;

abstract class AbstractRequest extends Request
{

    public function authorize()
    {
        return true;
    }
```

```
14
15      public function rules()
16      {
17          if ($this->isMethod('post') or $this->isMethod('put'))
18              return $this->rules;
19
20          return [];
21      }
22
23      public function messages()
24      {
25          return [
26              'required' => ':attribute não deve ficar vazio.',
27              'title.required' => 'O título é obrigatório',
28              'min' => ':attribute deve ter mais de :min caracteres.',
29              'numeric' => ':attribute deve ser um número.'
30          ];
31      }
32  }
```

Qualquer método pode ser sobrescrito, como `authorize()` por exemplo, ou deixar como informado acima, veja como ficaria a nossa classe `ProductsRequest`:

```
1   <?php
2
3   namespace App\Http\Requests;
4
5   use App\Http\Requests\AbstractRequest;
6
7   class ProductsRequest extends AbstractRequest
8   {
9
10      protected $rules = [
11          'title' => 'required|min:3',
12          'body' => 'required',
13          'value' => 'required|numeric',
14          'qtd' => 'required|numeric',
15      ];
16  }
```

E agora vamos deixar o IoC do Laravel trabalhar por nós, no seu `ProductsController` crie um método construtor:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\ProductsRequest;
use App\Http\Controllers\CrudController;
use Illuminate\Contracts\Validation\Validator;

class ProductsController extends CrudController
{
    protected $model = '\App\Product';
    protected $path = 'products';

    public function __construct(ProductsRequest $request)
    {

    }

}
```

Pode parecer errado um método vazio, afinal métodos não devem existir sem um propósito, mas ele tem um sim, o propósito de injetar o nosso Request personalizado para executar as validações, agora qualquer execução que for POST ou PUT vai validar os dados, inclusive no `update()` e `store()`.

Pode acontecer de você precisar validar dados fora do controller, neste caso você já tem a validação pronta para ser usada, essa é a real vantagem deste formato de validar dados.

Vamos ver como ficaria isso com o `Users`?

Request:

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\AbstractRequest;

class UsersRequest extends AbstractRequest
{
    protected $rules = [
        'name' => 'required|min:3',
        'email' => 'required|email'
    ];
}
```

Controller:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests\UsersRequest;
use App\Http\Controllers\CrudController;

class UsersController extends CrudController
{

    protected $model = '\App\User';
    protected $path = 'users';

    public function __construct(UsersRequest $request)
    {

    }
}
```

Que tal? Simples, rápido e eficiente, um CRUD completo em apenas algumas linhas e você pode personalizar o que quiser, apenas criando os métodos, o que vai substituir os originais.

Que tal você traduzir a mensagem de validação de emails?

Lembra daquele bloco de códigos que exibe as mensagens de erro, que tal criar um arquivo externo para ele, assim facilita as coisas pra gente.

O arquivo de view em `resources\views\helpers\validate_erros.blade.php` terá este conteúdo:

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

E nas views (por exemplo a `create.blade.php` de products)

```
<h1>Cadastrar</h1>

@include('helpers.validate_errors')

<form action="{{ route('products.store') }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        Title: <input type="text" name="title" value="{{ old('title') }}"><br>
        Description: <textarea name="body">{{ old('body') }}</textarea><br>
        Value: <input type="text" name="value" value="{{ old('value') }}"><br>
        Quantity: <input type="number" name="qtd" value="{{ old('qtd') }}"><br>
        Url: <input type="text" name="url" value="{{ old('url') }}"><br>
        <input type="submit">
</form>
```

Menos código espalhado pra gerenciar, não esqueça de aplicar nos demais formulários.

Relacionamentos

Agora precisamos relacionar nossos produtos com as categorias, mas antes, se você ainda não criou o cgerenciamento de categorias essa é uma excelente hora, vou deixar o código abaixo pra te ajudar, mas você já deve poder fazer isso sozinho.

Model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    protected $table = 'categories';
    protected $fillable = ['title'];
}
```

Controller:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\CategoriesRequest;
use App\Http\Controllers\CrudController;

class CategoriesController extends CrudController
{
    protected $model = '\App\Category';
    protected $path = 'categories';

    public function __construct(CategoriesRequest $request)
    {

    }

}
```

Request:

```php
<?php

namespace App\Http\Requests;

use App\Http\Requests\AbstractRequest;

class CategoriesRequest extends AbstractRequest
{
    protected $rules = [
        'title' => 'required|min:3',
    ];
}
```

View index:

```
<h1>Categories</h1>

<table>
        <thead>
                <tr>
                        <th>id</th>
                        <th>title</th>
                        <th>actions</th>
                </tr>
        </thead>
        <tbody>
        @foreach ($data as $k=>$v)
                <tr>
                        <td>{{ $k+1 }}</td>
                        <td>{{ $v->title }}</td>
                        <td>
                                <a href="{{ route('categories.show', ['id'=>$v->id]) }}">view</a>
                                <a href="{{ route('categories.edit', ['id'=>$v->id]) }}">edit</a>
                                <form action="{{ route('categories.update', ['id'=>$v->id]) }}" me
" style="display:inline-block">
                                        <input type="hidden" name="_token" value="{{ csrf_token()
                                        <input type="hidden" name="_method" value="DELETE">
                                        <input type="submit" value="remove">
                                </form>
                        </td>
```

```
                </tr>
        @endforeach
        </tbody>
</table>
```

View create:

```
<h1>Cadastrar</h1>

@include('helpers.validate_errors')

<form action="{{ route('categories.store') }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        Title: <input type="text" name="title" value="{{ old('title') }}"><br>
        <input type="submit">
</form>
```

View edit:

```
<h1>Editando {{ $data->name }}</h1>

@include('helpers.validate_errors')

<form action="{{ route('categories.update', ['id'=>$data->id]) }}" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        <input type="hidden" name="_method" value="PUT">
        Title: <input type="text" name="title" value="{{ $data->title }}"><br>
        <input type="submit">
</form>
```

View show:

```
<h1>{{ $data->title }}</h1>

<ul>
        <li>cadastro: {{ $data->created_at }}</li>
        <li>atualização: {{ $data->updated_at }}</li>
</ul>
```

E agora vamos ao que interessa, relacionamentos.

Relacionamentos no Eloquent são definidos dentro de métodos na model, imagine que cada produto pertence a várias categorias, e que o contrário também acontece, cada categoria pertence a vários produtos, este é um relacionamento muitos para muitos, é o que faremos aqui. Veja a seguir.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $table = 'products';

    /**
     * Permitir alterações em massa
     */
    protected $fillable = ['title', 'body', 'value', 'qtd', 'url'];

    public function setUrlAttribute($value)
    {
        if ($value=='')
            $value = $this->attributes['title'];

        $this->attributes['url'] = str_slug($value);
    }

        /**
         * Relacionamento entre o model Category e o atual (Product)
         */
    public function categories()
    {
            //este é um relacionamento muitos para muitos
        return $this->belongsToMany('App\Category');
    }
}
```

Relacionamentos sempre dão uma dor de cabeça danada aos desenvolvedores, mas o Laravel torna tudo muito simples, com apenas uma linha você já tem acesso a todos os dados em questão. Veja como resgatariamos os registros do model acima.

```
$product = \App\Product::find($id);
$categories = $product->categories;

echo '<h1>'. $product->title .'</h1>';

foreach ($categories as $category) {
        echo $category->title;
}
```

Simples não é, vamos mais a fundo e entender um pouco mais sobre o assunto.

Um usuário em uma rede social possui um único perfil, chamamos este relacionamento de **um para um**, por outro lado um usuário pode ter várias imagens no album, este relacionamente é **um para muitos**, a grande diferença entre **muitos para muitos** e **um para muitos** é que o primeiro é exatamente o mesmo se invertido, ao dento que a inversão de **um para muitos** é **um para um**, uma foto pertence a um usuário.

1 usuário tem 1 perfil -> 1 perfil pertence a 1 usuario -> um para um ou `has one`. 1 usuário tem muitas imagens -> 1 imagem pertence a 1 usuário -> muitos para um ou `has many` 1 produto tem muitas categorias -> 1 categoria tem muitos produtos -> muitos para muitos ou `many to many`.

O contrário de `has one` e `has many` é `belongs to` ou `pertence a`, o contrário de `many to many` é `many to many`, pegou o jeito? Vamos ao código então.

Para declarar um relacionamento `has one`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Retorna o registro do perfil associados ao usuário
     */
    public function profile()
    {
        return $this->hasOne('App\Profile');
    }
}
```

Para declarar um relacionamento `has many`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Retorna os registros de imagens associados ao usuário
     */
    public function images()
    {
        return $this->hasMany('App\Image');
    }
}
```

Para declarar um relacionamento `belongs to`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
    /**
     * Retorna o registro de usuário relacionado a imagem
     * Seria parecido com o perfil, mas no model Profile.
     */
    public function post()
    {
        return $this->belongsTo('App\User');
    }
}
```

Para declarar um relacionamento `many to many`, na verdade este é o nosso model já exibido antes.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $table = 'products';

    /**
     * Permitir alterações em massa
     */
    protected $fillable = ['title', 'body', 'value', 'qtd', 'url'];

    public function setUrlAttribute($value)
    {
        if ($value=='')
            $value = $this->attributes['title'];

        $this->attributes['url'] = str_slug($value);
    }

    public function categories()
    {
        return $this->belongsToMany('App\Category');
    }
}
```

Agora sempre que buscarmos um registro no banco teremos acesso as categorias e você ainda pode usar o query builder que estudamos no capítulo anterior, por exeplo, se tivessemos um campo `active` para informar se uma categoria está ativa ou não e ainda precisar ordernar pelo `title`:

```
$product = \App\Product::find($id);
$categories = $product->categories()->where('active', 1)->orderBy('title')->get(\
);
```

Agora imagine que você precise (e vamos precisar) de um array para um conjunto de checkboxes e assim podermos associar as produtos a categorias.

```
$product = \App\Product::find($id);
$checked = $product->categories->lists('title','id')->toArray();
```

Este código retornaria, por exemplo, o seguinte array.

```
[
        1=>'Informática',
        3=>'Eletrônicos',
        8=>'Notebooks'
];
```

O Laravel também facilita muito o processo de informar novos relacionamento entre registros com os métodos `attach()` para adicionar um relacionamento, o `detach()` para remover um relacionamento e o `sync()` que remove todos os relacionamentos e adiciona os novos informados, muito útil este último.

```
$product->categories()->attach(1); //cria um relacionamento
$product->categories()->detach(1); //remove um relacionamento
```

Ambos os métodos também aceitam um array com vários ids a relacionar ou remover o relacionamento.

```
$product->categories()->attach([1, 2, 3, 4, 5]);
```

E o método `attach()` ainda atualiza campos:

```
$product->categories()->attach(1, ['active'=>'s']); //atualiza e relaciona 1 reg\
istro
$product->categories()->attach([1=>'active'=>'s', 2, 3, 4, 5]); //atualiza e rel\
aciona vários registros
```

Além destes 2 ainda temos o `sync()` que remove todos os relacionamentos e relaciona somente os valores passados, ele aceita um array que pode ou não atualizar campos, assim como o `attach()`.

```
$product->categories()->sync([1=>'active'=>'n', 5]);
```

Com essas novas informações em mente vamos atualizar o nosso controller:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\ProductsRequest;
use App\Http\Controllers\CrudController;
use Illuminate\Contracts\Validation\Validator;

class ProductsController extends CrudController
{
    protected $model = '\App\Product';
    protected $path = 'products';

    public function __construct(ProductsRequest $request)
    {

    }

        /**
         * Novo action
         */
    public function categories(Request $request, $id)
    {
            $product = \App\Product::find($id);

            if ($request->isMethod('post')) {
                    $product->categories()->sync($request->input('categories'));
                    return redirect()->route($this->path.'.categories', ['id'=>$id]);
            }

            $categories = \DB::table('categories')->lists('title', 'id');
            $checked = $product->categories->lists('title','id')->toArray();

            return view (
                    $this->path.'.categories',
                    ['data'=>$product, 'checked'=>$checked, 'categories'=>$categories]
            );
    }

}
```

Também precisamos cria uma nova rota, já que esta nova action não será identificada pelo

`Route::resources()`, vamos usa o `Route::match()`.

```
Route::match(
        ['get', 'post'],
        '/products/categories/{id}',
        ['uses'=>'ProductsController@categories', 'as'=>'products.categories']
);
```

E finalmente a nossa view `categories.blade.php`.

```
<h1>Categorias de  {{ $data->title }}</h1>

<form action="{{ route('products.categories', ['id'=>$data->id]) }}" class="form\
" method="POST">
        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        <ul>
        @foreach ($categories as $k=>$category)
                <li>
                        <input type="checkbox" name="categories[]" value="{{$k}}" @if (!empty($che
d[$k])) checked @endif>
                        {{$category}}
                </li>
        @endforeach
        </ul>
        <input type="submit" class="btn btn-primary">
</form>
```

Mas não vai funcionar, ao enviar o formulário o `ProductsRequest` vai tentar validar os dados e, após não conseguir, irá redirecionar para o formulário novamente, não queremos que ele aplique as regras de validação aqui então vamos acertar nossa lógica e criar um método para verificar a action que está sendo chamada, isso no *AbstractRequest*.

```
protected function checkAction()
{
    if (empty($this->route()->getAction()['as']))
        return false;

    $base = explode('.', $this->route()->getAction()['as']);

    if (empty($base[1]))
        return false;
```

```
10
11      return in_array($base[1], $this->actionsToValidate);
12  }
```

Na primeira linha eu verifico se a rota atual é nomeada, se for eu quebro em duas partes com base no ponto (lembre-se que os nomes das rotas do *Route::resources()** seguem o padrão `controller.action`) e se existir mais de umitem no array eu uso o segundo elemento (o action) para verificar se ele está na lista de permissão definida no atributo `$actionsToValidate`.

Aplicaremos este novo método ao `rules()`, veja a classe completa.

```
1   <?php
2
3   namespace App\Http\Requests;
4
5   use App\Http\Requests\Request;
6
7   abstract class AbstractRequest extends Request
8   {
9
10      protected $actionsToValidate = ['store', 'update'];
11
12      public function authorize()
13      {
14          return true;
15      }
16
17      public function rules()
18      {
19          if (($this->isMethod('post') or $this->isMethod('put')) and $this->check\
20  Action()) {
21              return $this->rules;
22          }
23
24          return [];
25      }
26
27      public function messages()
28      {
29          return [
30              'required' => ':attribute não deve ficar vazio.',
31              'title.required' => 'O título é obrigatório',
32              'min' => ':attribute deve ter mais de :min caracteres.',
```

```
            'numeric' => ':attribute deve ser um número.',
            'email' => ':attribute deve ser um email válido.'
        ];
    }

    protected function checkAction()
    {
        if (empty($this->route()->getAction()['as']))
            return false;

        $base = explode('.', $this->route()->getAction()['as']);

        if (empty($base[1]))
            return false;

        return in_array($base[1], $this->actionsToValidate);
    }
}
```

Com isso podemos usar actions personalizadas e definir se vamos ou não valida-las substituindo o `$actionsToValidate` no controller original se necessário.

Falta só adicionar um link na listagem de produtos:

```
<a href="{{ route('products.categories', ['id'=>$v->id]) }}">relations</a>
```

Agora temos o *ProductsController* com um CRUD, validação e até relacionamento entre produtos e categorias, mas, embora funcional, ainda está feio o nosso painel, vamos resolver isso no próximo capítulo e ainda incrementar um sistema de autenticação.

Painel de administração

Tema da administração

Para o a administração da loja vou usar um tema construído Twitter Bootstrap, ele é gratuito e está disponível para download neste link: http://startbootstrap.com/template-overviews/sb-admin/[17], você pode baixar e descompactar em qualquer lugar FORA da instalação do Laravel.

Criando o tema

Precisamos que este tema (css, js, fontes, etc...) fique disponível para a internet, o Laravel protege todos os arquivos de acesso externo setando o **document root** no diretório public, em outras palavras, sempre que acessarmos o site no navegador estaremos, na verdade, acessando o diretório public da raiz do Laravel, isso evita que nossos scripts PHP e arquivos de configuração possam ser acessados.

Copie os 4 diretórios do tema para dentro de *public/admin* (crie se necessário) de forma que fique assim:

- public
  - admin
    * css
    * font-awesome
    * fonts
    * js

Com isso já podemos acessar o Twitter Bootstrap através do link *http://localhost:8000/admin/css/bootstrap.css*, por exemplo.

Dentro de *resources/views* vamos criar um novo arquivo chamado **admin.blade.php**, note que poderíamos criar direto na raiz do diretório *views*, mas vamos organizar um pouco as coisas, que tal um novo diretório para os temas? Mais interessante, não! Vou chamar de *layouts*, o caminho completo será **resources/views/layouts/admin.blade.php**, copie todo o código do arquivo **index.html** para o nosso layout admin.

Vou remover o conteúdo e separar a navegação (para ficar com arquivos menores e mais simples de alterar), remova tudo entre as linha 198 e 465 (esse seria o código dentro da div com a classe .container-fluid, mantenha esta div, só remova o conteúdo dela). Isso remove o conteúdo do tema, o Laravel vai se encarregar de inserir isso dinamicamente. No lugar do que código que removeu coloque apenas isso:

[17] http://startbootstrap.com/template-overviews/sb-admin/

```
@yield('content')
```

O `@yield()` informa um bloco de código ou string a ser inserido naquele ponto do tema, neste caso será o conteúdo que criamos para a administração, ou seja, as tabelas, formulários e tudo o que criamos nos capítulos anteriores.

Agora copie tudo da linha 40 até a 193 (.navbar-collapse) e cole tudo em um novo arquivo em **resources/views/layouts/navigation/admin.blade.php** (você terá que criar este arquivo), no lugar do que copiou de **admin.blade.php** (linhas 40 a 193) cole o código abaixo (sim, substitua tudo).

```
@include('layouts.navigation.admin')
```

Isso finaliza as coisas por enquanto.

O `@include()` funciona como o `include()` do PHP, ou seja, carrega um arquivo externo.

O nosso tema `admin` deve estar como o código abaixo.

```
<!DOCTYPE html>
<html lang="pt-br">

<head>

    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Loja Laravel</title>

    <link href="/admin/css/bootstrap.min.css" rel="stylesheet">
    <link href="/admin/css/sb-admin.css" rel="stylesheet">
    <link href="/admin/css/plugins/morris.css" rel="stylesheet">
    <link href="/admin/font-awesome/css/font-awesome.min.css" rel="stylesheet" t\
ype="text/css">

    <!--[if lt IE 9]>
        <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"><\
/script>
        <script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js\
"></script>
    <![endif]-->

</head>
```

```
26
27  <body>
28
29      <div id="wrapper">
30
31          @include('layouts.navigation.admin')
32
33          <div id="page-wrapper">
34
35              <div class="container-fluid">@yield('content')</div>
36              <!-- /.container-fluid -->
37
38          </div>
39          <!-- /#page-wrapper -->
40
41      </div>
42      <!-- /#wrapper -->
43
44      <!-- jQuery -->
45      <script src="/admin/js/jquery.js"></script>
46      <script src="/admin/js/bootstrap.min.js"></script>
47      <script src="/admin/js/plugins/morris/raphael.min.js"></script>
48      <script src="/admin/js/plugins/morris/morris.min.js"></script>
49      <script src="/admin/js/plugins/morris/morris-data.js"></script>
50
51  </body>
52
53  </html>
```

Vamos trabalhar naquele menu de navegação? Eu removi a sessão *Top Menu Items* completamente e criei links para os diversos cruds que criamos, ficou assim:

```
1   <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">
2       <div class="navbar-header">
3           <button type="button" class="navbar-toggle" data-toggle="collapse" data-\
4   target=".navbar-ex1-collapse">
5               <span class="sr-only">Toggle navigation</span>
6               <span class="icon-bar"></span>
7               <span class="icon-bar"></span>
8               <span class="icon-bar"></span>
9           </button>
10          <a class="navbar-brand" href="index.html">Administração</a>
```

```
11      </div>
12      <div class="collapse navbar-collapse navbar-ex1-collapse">
13          <ul class="nav navbar-nav side-nav">
14              <li>
15                  <a href="{{ route('products.index') }}"><i class="fa fa-fw fa-da\
16  shboard"></i> Produtos</a>
17              </li>
18              <li>
19                  <a href="{{ route('categories.index') }}"><i class="fa fa-fw fa-\
20  dashboard"></i> Categorias</a>
21              </li>
22              <li>
23                  <a href="{{ route('users.index') }}"><i class="fa fa-fw fa-dashb\
24  oard"></i> Usuários</a>
25              </li>
26          </ul>
27      </div>
28  </nav>
```

Você lembra do `route()`? Usei ele nos links para gerar uma url com base em uma rota nomeada.

Usando o tema

Agora precisamos informar que as nossas views devem usar o tema que criamos, lembra da view **products.index** (você já deve saber que ela fica em *resources/views/products/index.blade.php), precisamos que ela siga a regra a seguir:

```
1  @extends('layouts.admin')
2
3  @section('content')
4      //conteudo da action
5  @endsection
```

O `@extends` informa o layout a usar (neste caso, o que acabamos de criar) e o `@section` o conteúdo a se colocar em um `@yield` do tema, criamos um `@yield('content')`, então o `@section('content')` informa o que deve ser colocado naquele `@yield()`, o resultado final do **products.index**:

```
@extends('layouts.admin')

@section('content')
    <h1 class="page-header">Products</h1>

        <table class="table table-hover table-striped">
                <thead>
                        <tr>
                                <th>id</th>
                                <th>title</th>
                                <th>qtd</th>
                                <th class="text-right">actions</th>
                        </tr>
                </thead>
                <tbody>
                @foreach ($data as $k=>$v)
                        <tr>
                                <td>{{ $k+1 }}</td>
                                <td>{{ $v->title }}</td>
                                <td>{{ $v->qtd }}</td>
                                <td class="text-right">
                                        <a href="{{ route('products.show', ['id'=>$v->id]) }}" cla
ry btn-xs">view</a>
                                        <a href="{{ route('products.edit', ['id'=>$v->id]) }}" cla
lt btn-xs">edit</a>
                                        <a href="{{ route('products.categories', ['id'=>$v->id])
-default btn-xs">relations</a>
                                        <form action="{{ route('products.update', ['id'=>$v->id])
method="POST" style="display:inline-block">
                                                <input type="hidden" name="_token" value="{{ csrf_
                                                <input type="hidden" name="_method" value="DELETE"
                                                <input type="submit" value="remove" class="btn btn
                                        </form>
                                </td>
                        </tr>
                @endforeach
                </tbody>
        </table>
@endsection
```

Eu aproveitei para incluir algumas classes css do Twitter Bootstrap, o mesmo pode ser feito em todas as outras views.

Rota com /admin

Agora vamos adicionar um `/admin` nas nossas URLs, isso é simples, basta agruparmos as rotas abaixo.

```
Route::match(
        ['get', 'post'],
        '/products/categories/{id}',
        ['uses'=>'ProductsController@categories', 'as'=>'products.categories']
);

Route::resource('products', 'ProductsController');
Route::resource('users', 'UsersController');
Route::resource('categories', 'CategoriesController');
```

Colocando dentro de:

```
Route::group(['prefix' => 'admin'], function () {
    //rotas
});
```

O resultado final:

```
Route::group(['prefix' => 'admin'], function () {
    Route::match(
                ['get', 'post'],
                '/products/categories/{id}',
                ['uses'=>'ProductsController@categories', 'as'=>'products.categories']
        );

        Route::resource('products', 'ProductsController');
        Route::resource('users', 'UsersController');
        Route::resource('categories', 'CategoriesController');
});
```

Tem um detalhe, todos os nomes de rotas que o `Route::resource` cria recebe um `admin.` por conta do `prefix` que usamos, por exemplo, a `products.index` mudou para `admin.products.index`, você deve receber um erro ao acessar a administração.

Embora essa mudança não programada possa parecer problemático para a maioria, isso te ajuda a organizar melhor sua aplicação quando ela tem várias "sessões" ou "áreas", imagine o WebDevBr[18],

[18] http://www.webdevbr.com.br

eu tenho um CRUD de cursos na administração e um CRUD de cursos para o professor, outro controller cursos para a loja e outro para a área do aluno, cada um com suas validações e models diferentes (sim, tenho regras diferentes em todos os casos), poderia diminuir a quantidade de controllers, mas classes menores geram código mais simples de manter (não exagere), veja como ficam as minhas rotas.

- admin.courses.index
- professor.courses.index

Veja como fica a nossa view *users/index.blade.php* com as rotas corrigidas.

```
@extends('layouts.admin')

@section('content')
        <h1 class="page-header">
            Usuários
            <small><a href="{{ route('admin.users.create') }}" class="btn btn-success b\
tn-xs">novo</a></small>
    </h1>

        <table class="table table-hover table-striped">
                <thead>
                        <tr>
                                <th>id</th>
                                <th>name</th>
                                <th>mail</th>
                                <th class="text-right">actions</th>
                        </tr>
                </thead>
                <tbody>
                @foreach ($data as $k=>$v)
                        <tr>
                                <td>{{ $k+1 }}</td>
                                <td>{{ $v->name }}</td>
                                <td>{{ $v->email }}</td>
                                <td class="text-right">
                                        <a href="{{ route('admin.users.show', ['id'=>$v->id]) }}"
imary btn-xs">view</a>
                                        <a href="{{ route('admin.users.edit', ['id'=>$v->id]) }}"
fault btn-xs">edit</a>
                                        <form action="{{ route('admin.users.update', ['id'=>$v->i
m" method="POST" style="display:inline-block">
```

```
                                                    <input type="hidden" name="_token" value="{{ csrf_
                                                    <input type="hidden" name="_method" value="DELETE"
                                                    <input type="submit" value="remove" class="btn btn
                                            </form>
                                    </td>
                            </tr>
                    @endforeach
                    </tbody>
            </table>
@endsection
```

E o nosso `CrudController` também recebe o `admin.` em todos os redirects:

```
return redirect()->route('admin.'.$this->path.'.index');
```

Outra vantagem em agrupar rotas é a facilidade que temos em trabalhar com namespaces, sub-domínios e middleware, tudo o que é configurado no grupo reflete automaticamente em todas as rotas dentro dele.

Com grupos de rotas eu posso organizar os controllers, estes que criamos são para a administração, não quero usar os mesmos para a loja virtual, quanto menor e mais focada as classes melhor para a manutenção.

A melhor forma de fazer isso é adicionando um sub-namespace, isso irá, consequentemente, adicionar um novo diretório também, o sub-namespace será o *Admin*.

```
Route::group(['prefix' => 'admin', 'namespace' => 'Admin'], function () {
    Route::match(
                ['get', 'post'],
                '/products/categories/{id}',
                ['uses'=>'ProductsController@categories', 'as'=>'products.categories']
        );

        Route::resource('products', 'ProductsController');
        Route::resource('users', 'UsersController');
        Route::resource('categories', 'CategoriesController');
});
```

Os controllers `CategoriesController`, `ProductsController` e `UsersController` agora recebem um novo sub-namespace e devem ser movidos para o novo diretório *app/Http/Controllers/Admin*, neste ponto você pode receber um erro informando que a classe `CrudController` não foi encontrada, é só informar no `use`, deve ficar assim:

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\CrudController;
```

Agora sim, tudo organizado!

Para ter um panorama mais detalhado deste passo você pode consultar este capítulo da documentação (em inglês) http://laravel.com/docs/5.1/routing#route-groups[19].

Middleware

> Middleware ou mediador, no campo da computação distribuída, é um programa de computador que faz a mediação entre software e demais aplicações. É utilizado para mover ou transportar informações e dados entre programas de diferentes protocolos de comunicação, plataformas e dependências do sistema operacional. – *Wikipedia*

No Laravel, os middleware interceptam a requisição HTTP antes de ser enviado para o controller e com isso podemos filtrar ou tomar uma decisão antes que qualquer outra coisa seja feita e isso é fantástico, já que podemos usar o middleware de autenticação (que já está pronto) para permitir o acesso ou redirecionar para a página de login.

Além da autenticação o Laravel já vem com alguns middlewares prontos para serem usados e disponíveis em app/Http/Middleware, citando a documentação:

> Existem vários middlewares incluídos no Laravel Framework, incluindo middleware para manutenção, autenticação, proteção CSRF, e muito mais. Todos estes middleware estão localizados no diretório app/Http/Middleware.

Agora que você entendeu o que é um middleware vamos construir nossa autenticação.

Para ter um panorama mais detalhado deste passo você pode consultar este capítulo da documentação (em inglês) http://laravel.com/docs/5.1/middleware[20].

Como configurar a autenticação

Para começar crie as rotas de autenticação fora do nosso grupo.

[19] http://laravel.com/docs/5.1/routing#route-groups
[20] http://laravel.com/docs/5.1/middleware

```
Route::get('admin/auth/login', 'Auth\AuthController@getLogin');
Route::post('admin/auth/login', 'Auth\AuthController@postLogin');
Route::get('admin/auth/logout', 'Auth\AuthController@getLogout');
```

O controller `Auth\AuthController` já vem pronto junto com o Laravel, mas a views não, então crie um diretório de view chamado `auth` e dentro um arquivo chamado `login.blade.php` com este conteúdo:

```
@extends('layouts.admin')

@section('content')
    <h1 class="page-header text-center">Autenticação</h1>
    <form method="POST" action="/admin/auth/login" class="col-md-4 col-md-offset\
-4">
        {!! csrf_field() !!}
        <div>
            Email
            <input type="email" name="email" value="{{ old('email') }}" class="f\
orm-control">
        </div>
        <div>
            Password
            <input type="password" name="password" id="password" class="form-con\
trol">
        </div>
        <div>
            <input type="checkbox" name="remember"> Remember Me
        </div>
        <div>
            <button type="submit" class="btn btn-primary">Login</button>
        </div>
    </form>
@endsection
```

Quase pronto, agora precisamos proteger as páginas, para isso vamos até o nosso grupo de rotas admin e adicionar o middleware:

```
Route::group(['prefix' => 'admin', 'namespace' => 'Admin', 'middleware'=>'auth']\
, function () {
    Route::match(
                ['get', 'post'],
                '/products/categories/{id}',
                ['uses'=>'ProductsController@categories', 'as'=>'products.categories']
        );

        Route::resource('products', 'ProductsController');
        Route::resource('users', 'UsersController');
        Route::resource('categories', 'CategoriesController');
        Route::get('auth/login', 'Auth\AuthController@getLogin');
        Route::post('auth/login', 'Auth\AuthController@postLogin');
        Route::get('auth/logout', 'Auth\AuthController@getLogout');
});
```

Agora quando tentarmos acessar qualquer página da administração seremos redirecionados para a `/auth/login` e epa! Como assim? Não era `/admin/auth/login`, cade o *admin* na rota, precisamos acertar isso. Vá até o middleware `Authenticate` em *app/Http/Middleware/Authenticate.php* e no método `handle()`, altere no else (linha 41, aproximadamente) de:

```
return redirect()->guest('auth/login');
```

Para:

```
return redirect()->guest('admin/auth/login');
```

Agora fomos para o local certo, mas com um erro de loop, para resolver isso apenas faça uma verificação de rota:

```
public function handle($request, Closure $next)
{
    if ($this->auth->guest()) {
        if ($request->ajax()) {
            return response('Unauthorized.', 401);
        } else if ($request->path() != 'admin/auth/login') {
            return redirect()->guest('admin/auth/login');
        }
    }

    return $next($request);
}
```

Preste atenção ali no `else if`.

Agora no controller `AuthController` vamos adicionar dois atributos.

```
1 protected $loginPath = '/admin/auth/login'; //endereço de login
2 protected $redirectPath = '/admin/products'; //endereço de redirecionamento após\
3  o login
```

Pronto, agora você deve ser capaz de acessar com seu email e senha, para deslogar acesse */admin/auth/logout*.

Para ter um panorama mais detalhado deste passo você pode consultar este capítulo da documentação (em inglês) http://laravel.com/docs/5.1/authentication[21].

[21] http://laravel.com/docs/5.1/authentication

Site

Tema da loja

Em breve.

Listagem de categorias com View Composer

Em breve.

Listagem de produtos por categorias

Em breve.

Página de produtos

Em breve.

Carrinho de compras

Criando model sem acesso a banco de dados

Em breve.

Adicionar produto

Em breve.

Remover produto

Em breve.

Alterar quantidade de um produto

Em breve.

Listar no carrinho de compras

Em breve.

Finalizar compra com registro ou login do usuário

Em breve.

Integração com Web Services

Integrando com o PagSeguro

Em breve.

Integrando com os Correios

Em breve.